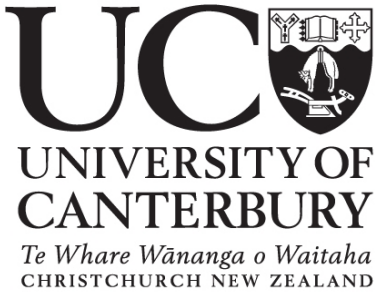
UNIVERSITY OF CANTERBURY

Department of Physics and Astronomy

CHRISTCHURCH NEW ZEALAND




# Evolving Turing's Artificial Neural Networks

by

Ewan Orr

Doctor of Philosophy Thesis

2010

# Abstract

Our project uses ideas first presented by Alan Turing. Turing's immense contribution to mathematics and computer science is widely known, but his pioneering work in artificial intelligence is relatively unknown. In the late 1940s Turing introduced discrete Boolean artificial neural networks and, it has been argued that, he suggested that these networks be trained via evolutionary algorithms. Both artificial neural networks and evolutionary algorithms are active fields of research. Turing's networks are very basic yet capable of complex tasks such as processing sequential input; consequently, they are an excellent model for investigating the application of evolutionary algorithms to artificial neural networks.

We define an example of these networks using sequential input and output, and we devise evolutionary algorithms that train these networks. Our networks are discrete Boolean networks where every 'neuron' either performs NAND or identity, and they can represent any function that maps one sequence of bit strings to another. Our algorithms use supervised learning to discover networks that represent such functions. That is, when searching for a network that represents a particular function our algorithms use input-output pairs of that function as examples to aid the discovery of solution networks.

To test our ideas we encode our networks and implement the algorithms in a computer program. Using this program we investigate the performance of our networks and algorithms on simple problems such as searching for networks that realize the parity function and the multiplexer function. This investigation includes the construction and testing of an intricate crossover operator. Because our networks are composed of simple 'neurons' they are a suitable test-bed for novel training schemes.

To improve our evolutionary algorithms for some problems we employ the symmetry of the problem to reduce its search space. We devise and test a means of using subgroups of the group of permutation of inputs of a function to aid evolutionary searches search for networks that represent that function. In particular, we employ the action of the permutation group $S_2$ to 'cut down' the search space when we search for networks that represent functions such as parity.

# Acknowledgements

I am indebted to many people who helped me complete this thesis. I try to acknowledge these people with the following unordered list.

Dr Ben Martin for investing a great amount of time into this project. For mentoring me, teaching me, and for expert problem solving. Thank you.

Dr William Joyce for investing a great amount of time into me in the early stages of my doctroal program. Furthermore, for his courage to consider big ideas and tackle difficult problems.

Prof Jack Copeland for introducing me to Turing's A-type networks as a workable test-bed for our ideas. Also, for discussions of these ideas.

Assoc Prof Michael Reid for giving me the freedom to pursue my interests.

University of Canterbury's Department of Physics and Astronomy for their support, including financial support.

Dr Brent Martin for answering the occasional question and letting me sit in on his machine learning course.

Dr Warwick Irwin for helping me with some design patterns questions.

Dr Marie Hale and Dr Tammy Steeves for letting me sit in on a course on evolution.

Richard Graham for computer support and helping me with numerous computer science queries.

Peter Smale for helping me with shell scripting.

Dr Niels Gresnigt, André Geldenhuis, Nishanthan Rabeendran, Pubudu Senanayake, and Sebastian Horvath for their help proof-reading this thesis.

Dr Orlon Petterson and Paul Brouwers for giving me access to many computers.

Throughout this project I have received support and assistance from many people. There is a fair chance that I have omitted deserving people from these acknowledgments and in advance I apologise for this oversight. I am grateful for all of the help that I received while I pursued this interesting project.

# Contents

# Symbols and Acronyms

| | |
|---|---|
| $\mathbb{N}_0$ | The denotes the non-negative integers. |
| $\mathbb{Z}_2$ | The set $\{0, 1\}$. We use this to describe the set of possible values of a Boolean variable. |
| $[a, b]$ | The set of real numbers between $a$ and $b$ inclusive. |
| $X^n$ | The set of ordered $n$-tuples, each entry of each $n$-tuple is an element of the set $X$. For example, if $X = \mathbb{Z}_2$ then $(0, 1, 1)$ and $(1, 0, 1)$ are two distinct members of $(\mathbb{Z}_2)^3$. |
| $\overline{\wedge}$ | The NAND operator. For example, consider the two Boolean variables $A$ and $B$. We denote the NAND of these two variables by $A \mathbin{\overline{\wedge}} B$. Often NAND is denoted by $\overline{A.B}$. We adopt the former convention to help clarify large bracketed NAND expressions. |
| $M[m, n]$ | The set of all $m \times n$ matrices with every matrix entry an element of $\mathbb{Z}_2$. |
| input-output pair | Consider some function $f : X \to Y$. When we write about an input-output pair of $f$ we are referring to the pair $(x, f(x))$ for some $x \in X$. |
| AI | artificial intelligence |
| ANN | artificial neural network |
| BFSM | Boolean finite state machine |
| EA | evolutionary algorithm |
| EC | evolutionary computing |
| FSM | finite state machine |
| ML | machine learning |

# 1 Introduction

## 1.1 Aims of this Chapter

In this chapter we present our motivation for this project. Also, we present a chapter by chapter overview.

## 1.2 Motivation

Our project considers very basic artificial neural networks (ANNs) and investigates using evolutionary algorithms (EAs) to train these networks. We have five points of motivation for this project. First, this project investigates historically interesting ideas. Second, this project develops very simple ANNs that accept and return sequential data. Third, this project employs EAs and investigates whether we can discover biologically analogous evolutionary operators for our ANNs. Fourth, this project investigates the use of symmetry to improve some machine learning (ML) techniques. Fifth, this project investigates techniques that intersect with automated scientific discovery—a topic of great interest to the author. Next we elaborate on each of these points.

### 1.2.1 Historical Importance

Alan Turing's contribution to mathematics and computer science is considerable. Also, Turing investigated topics that remain active areas of artificial intelligence research. In 1948 Turing wrote the paper *Intelligent Machinery* [1]. In this pioneering paper Turing introduced a type of ANN and he suggested that it be trained with EAs (see Section 3.7). Turing's ANNs, which are called A-types, are composed of basic and identical neurons each of which performs the Boolean operation NAND. Because every Boolean expression can be rewritten as an expression with only NAND operators A-types can represent any Boolean function. The simplicity of A-types make them an excellent test-bed for new ideas. However, in spite of their historical significance and their utility, A-types have received relatively little attention.

In this project we interpret Turing's A-types so that they can accept sequential input and return sequential output. The most notable continuation of research into Turing's networks is research conducted by Teuscher [2]. Teuscher experiments with A-types with fixed input states; for instance, he uses A-types in this manner to solve basic pattern classification tasks and shows that their dynamics are analogous to a non-linear oscillator [3]. Turing's definition

does not explicitly specify how A-types can be implemented as machines that accept input strings and return output strings. We invest considerable effort making our interpretation of Turing's A-types precise. We adopt language from graph theory to help explain our approach. We also introduce a second type of basic neuron to our A-type networks. This neuron enables the synchronization of information that flows through the A-type. We argue that the introduction of this second type of neuron is necessary for our interpretation of A-types. Furthermore, we provide experimental evidence that supports this claim. When we define an A-type we employ the definition of a finite state machine to aid precision. This is useful because we require careful consideration of the discrete and dynamic nature of our A-types.

### 1.2.2 Artificial Neural Networks

The mammalian brain is the most successful problem solver known. This is strong motivation for even loosely analogous techniques in ML. We understand that the architecture of the mammalian brain is a staggering number of interconnected relatively simple elements (the human brain contains approximately $7 \times 10^{10}$ neurons [4, p733]). The connections between these elements are adjusted as a result of learning. Analogously, ANNs contain interconnected simple elements whose interconnections are adjusted as they are trained for a particular task. Many successful modern implementations of ANNs ignore the timing within the networks. This makes their implementation relatively straightforward; however, in such implementations the biological analogy is rather tenuous.

Our interpretation of Turing's A-types give dynamic neural networks that accept and return sequential data. Although our networks are discrete our simple networks have time dependence; as do biological neural networks. We implement our A-types in a computer program; this enables us to experiment with novel training schemes for these networks.

### 1.2.3 Evolutionary Algorithms

The staggering complexity of life—including the complexity of the mammalian brain—is a consequence of evolution. This motivates analogous ML techniques which are broadly classed as EAs. In this project we devise EAs to train populations of A-types. We implement this in a computer program (we outline this program in Appendix A). Using this program we investigate the performance of our algorithms on a few simple problems. In these experiments we find that our EA significantly outperforms a blind search.

It is not clear whether recombination operators are beneficial in EAs. Although EAs are inspired by biology, links between the two fields are often tenuous. Biological chromosomes are immensely more complex than the analogous data structures of an EA. Similarly, most variation operators in biology, such as crossover, are usually incredibly sophisticated in comparison to those in an EA. Whether crossover is useful in an EA is currently an active area

of research [5] [6] [7]. In this project we attempt to construct a crossover operator that performs better than a macro-mutation operator. We employ ideas from graph theory to aid this attempt. We implement this in our computer program and test it via simulation. Finally, we speculate that a more worthwhile approach is to extend the evolution analogy to discover useful crossover operators.

Some researchers argue that sexual selection is responsible for human intelligence. This rather speculative idea* provides further motivation to employ crossover to train ANNs. The cognitive psychologist Geoffrey Miller advocates that sexual selection exerts an evolutionary pressure that results in human level intelligence [9]. Our seemingly redundant intellect is likened to the peacock's tail feathers. So if Miller is correct then evolution that uses sexual recombination can return a useful neural network, namely the human brain. With this in mind, we speculate that an EA that uses sexual recombination is an appropriate search method for artificial neural networks. At least it may provide a means of examining the link between cognition and computation.

### 1.2.4 Making Use of Symmetry

Symmetry, the notion of which is made precise by group theory, is an important concept in science. For instance, group theory is a pivotal part of modern particle physics [10]. Group theory is rather abstract but its application leads to useful problem solving techniques. The utility of abstract mathematics is also applied to ML [11]. Recently Kondor [12] investigated the use of group theoretic methods to improve some modern ML techniques. Kondor introduces this work as having two main themes: learning on domains that have non-trivial algebraic structure; and learning in the presence of invariances [12, p6]. We pursue this idea by using group theory to cut-down the A-type hypothesis space of our EAs. That is, we use the symmetry of a problem as background knowledge to decrease the hypothesis space of a search for a solution to that problem. Other researchers also apply symmetries to ANNS for this purpose [13]. We apply group theoretic notions to ANNs in a manner similar to that proposed by Shawe-Taylor [14] [15]. However, we permit feedback (Shawe-Taylor's networks are feedforward; that is, their directed graphs have no closed paths). Furthermore, we implement our scheme with an EA. Recently Dong and Zhang [16] incorporated group theoretic techniques into EAs with populations of ANNs. Their approach employs relatively simple operations on lists that represent ANNs. For our research the author's co-supervisor Dr Ben Martin devised an algorithm that evolved (A-type, symmetry) pairs. This algorithm implements intricate mutation operators that preserve a A-type's symmetry. To our knowledge our implementation is new and these methods have not been applied to A-types.

*Note that there are several competing theories of the evolution of intelligence. Most notably the Machiavellian intelligence hypothesis [8]. This hypothesis suggests that the complexities of social interaction is responsible for human intelligence.

### 1.2.5 Automating Scientific Discovery

Džeroski *et al.* provide a succinct definition of automating scientific discovery: it is the field of research that enables computers to produce results, that had a human scientist done the same, would be referred to as scientific discoveries [17, p1]. This research has a long history with some prominent proponents; for instance, Francis Bacon claimed that scientific theories could be generated from observations by a mechanical process [18, p5]. In the 1980's Simon *et al.* developed the computer program BACON which reproduced historically important discoveries such as Kepler's third law [19]. Automated techniques have discovered new scientific laws in protein folding [20][18, ch.2], they are used to discover elementary particles [21], and the *robot scientist* project aims to entirely automate yeast gene discovery [22] [23]. An excellent modern survey the topic is given in [17].

The immense goal of automating scientific discovery is the primary motivation for this project. This subject receives little mention in this thesis because, although it is of great interest—and motivation—to the author, our research is rather specialized. The author's goal was to employ symmetry ideas to aid automating scientific discovery. These lofty ideas evolved into the concrete—and achievable—task of investigating discrete ANNs trained with EAs.

Both ANNs and EAs are current research tools for automated scientific discovery. Also, using groups of symmetries to aid scientific discovery is an established technique [24]. Our project is a fundamental investigation into topics that intersect with this research.

## 1.3 Chapter Overview

Chapter 2 presents the mathematics, notation, and conventions that we adopt for this thesis. We present basic notions from graph theory, how we describe bit strings, and how we describe algorithms.

Chapter 3 presents a biased overview of ML. We discuss evolutionary computing, ANNs, and the A-type ANNs that Alan Turing invented.

Chapter 4 presents a formal definition of finite state machines, and uses this to define our A-types. We define A-types with three types of nodes: input nodes, nand machines, and delay machines.

Chapter 5 presents a definition of how an A-type can accept and output data packets. This definition makes precise our A-types' operation when they accept and output sequential data. Also in Chapter 5 we define how an A-type can represent a Boolean function. Furthermore, we discuss why we define A-types with delay machines.

Chapter 6 details EAs with populations of A-types. In particular, we detail one EA that has crossover, and two special cases: a mutation-only EA, and a blind search. When detailing these algorithms we describe how we construct a random A-type, how we assess an A-type's

fitness, how we construct a mutant A-type, and how we construct an A-type via crossover.

Chapter 7 details the computer simulations that we conducted using A-types. First we investigate our claim that A-types require delay machines to represent functions with sequential input and output. Second, we compare the performance of our blind search, mutation-only EA, and EA with crossover on four benchmark problems: these are searches for A-types that represent clamped $n$-identity, clamped $n$-parity, clamped $n$-multiplexer, and $n$-carry (a basic sequential function that we contrived). Third, we investigate whether our crossover operator is acting simply as a macro-mutator.

Chapter 8 details our attempt to employ the notion of symmetry to improve our EAs. We make this precise by employing group theory. We consider problems that are invariant under permutations of input variables; that is, invariant under the action of some subgroup of the symmetry group $S_n$ (where $n$ is the number of variables of the problem). We devise, and test via computer simulation, two schemes to employ symmetry with A-types. First we investigate a 'top-down' approach: this is a mutation-only EA where an A-type's fitness depends, in part, on an estimate of that A-type's invariance under permutations of input nodes. Second, we investigate a 'bottom-up' approach: this is a mutation-only EA where every A-type in the initial population has a symmetry of the problem, and every mutation preserves the original A-type's symmetry.

Chapter 9 gives a summary of the main results of this project. Also, we suggest future research that could stem from this work.

# 2 Mathematical Preliminaries

## 2.1 Aims of this Chapter

To make this thesis precise we employ some mathematical language. In particular, when we describe networks and manipulations of these networks we employ graph theory. We have chosen to present the requisite mathematics here rather than dispersing it throughout the thesis. When mathematics is required in later chapters we refer the reader to this chapter and to the literature. For the mathematically advanced reader this chapter may solely serve as an explanation of the notation that we have adopted.

## 2.2 Graphs

The notion of a graph and, in particular, the notion of a directed graph offers a means of precisely describing how information can 'percolate through' a network*. For much of the material in this section we borrow heavily from Diestel [26, chap 2] and Cormen *et al.* [27, chap 5].

### Simple Graphs

Informally, a graph consists of vertices and edges. Each edge is associated with two distinct vertices, and a pair of vertices has at most one edge associated with it. The following definition formally presents this idea [26, p2].

**Definition 2.1** (Graph)**.** A *graph*, $G$, is a pair $(V, E)$ of disjoint sets (*vertices* and *edges*) satisfying $E \subseteq V^2$. An edge $e$ and a vertex $v$ are *incident* if $v \in e$. The two vertices incident with an edge $e$ are called $e$'s *endpoints*. The number of edges incident to a vertex $v$ is the *degree* of $v$.

□

For example, consider the graph $G = (V, E)$, where $V = \{1, 2, 3, 4\}$ and $E = \{\{1, 3\}, \{2, 3\}, \{3, 4\}\}$. We illustrate this graph in Figure 2.1. This shows that for some graphs a diagram is a quick and easy means of accurately representing that graph.

*See Bose and Liang [25] for a reference that makes significant use of graph theory to describe artificial neural networks (ANNs) and their operation—we introduce ANNs in Chapter 3.

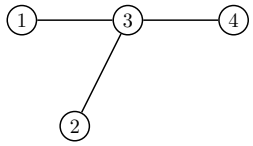



**Figure 2.1:** An example of a graph.

Note that we have defined a graph such that an edge $\{v, w\}$ must be the only edge between the vertices $v$ and $w$, and $v \neq w$. Some authors define graphs that allow multiple edges between two nodes. Such graphs are called *multigraphs* and, when the distinction is necessary, our graphs are called *simple graphs*.

### Directed Graphs

Now we introduce directed graphs. Informally, a directed graph consists of vertices and arrows, every arrow begins on a vertex and ends on a vertex. The following definition formally presents this idea. For this definition we borrow heavily from Diestel [26, p25].

**Definition 2.2** (Directed Graph)**.** A *directed graph*, $G$, is a pair $(V, E)$ of disjoint sets (*vertices* and *edges*) together with two maps $init : E(G) \longrightarrow V(G)$ and $term : E(G) \longrightarrow V(G)$ assigning to every edge $e \in E(G)$ an *initial vertex* $\text{init}(e)$ and a *terminal vertex* $\text{term}(e)$. The edge $e$ is said to *exit* from $\text{init}(e)$ and *enter* into $\text{term}(e)$. The *indegree* of a vertex is the number of edges entering that vertex. Similarly, the *out-degree* of a vertex is the number of edges exiting that vertex. If $\text{init}(e) = \text{term}(e)$, then the edge $e$ is called a *loop*.

□

For example, consider the directed graph $D = (V, E, init, term)$ where
$V = \{1, 2, 3, 4, 5\}$, $E = \{(1,2), (2,1), (2,2), (2,5), (2,5), (3,4), (4,5), (5,3)\}$,

$init : E \longrightarrow V$ is defined by

$$\begin{aligned} init(1,2) &= 1 \\ init(2,1) &= 2 \\ init(2,2) &= 2 \\ init(2,5) &= 2 \\ init(3,4) &= 3 \\ init(4,5) &= 4 \\ init(5,3) &= 5 \end{aligned}$$

and $term : E \longrightarrow V$ is defined by

$$\begin{aligned} init(1,2) &= 1 \\ init(2,1) &= 2 \\ init(2,2) &= 2 \\ init(2,5) &= 2 \\ init(3,4) &= 3 \\ init(4,5) &= 4 \\ init(5,3) &= 5 \end{aligned}$$

We illustrate this graph in figure 2.2. Usually a diagram is a quick and easy means of accurately representing a directed graph—as is often the case with a simple graph. We see that the edge $(1, 2)$ exits vertex 1 and enters vertex 2; whereas, the edge $(2, 1)$ exits vertex 2

and enters vertex 1. We see that vertex 5 has an indegree of 3 and an outdegree of 1. Also, the edge $(2, 2)$ is a loop.

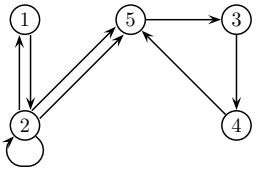



**Figure 2.2:** An example of a directed graph.

Now we consider further detail of directed graphs.

### Subgraph

Given a directed graph $G$ we can generate other directed graphs by considering subsets of $G$'s vertex set and subsets of $G$'s edge set. We formalize this in the following definition.

**Definition 2.3** (Subgraph)**.** The directed graph $G' = (V', E', init', term')$ is a *subgraph* of the directed graph $G = (V, E, init, term)$ if $V' \subseteq V$, $E' \subseteq E$, $init' = init$ when the domain of $init$ is restricted to that of $init'$, and $term' = term$ when the domain of $term$ is restricted to that of $term'$. The subgraph of $G$ *induced* by $V'$ is the graph $G' = (V', E', init', term')$ where $E'$ is the set of all edges in $G$ between nodes in $V'$ and $init' = init$ with the domain of $init$ is restricted to $E'$, and $term' = term$ with the domain of $term$ is restricted to $E'$.

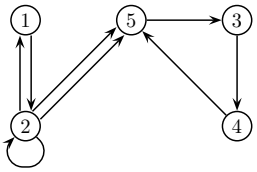
□

For example, Figure 2.3 shows a directed graph and some of its subgraphs.

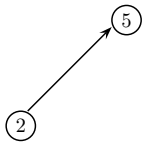
### Connectedness

If we traverse a directed graph by 'moving only along the direction of arrows' and in doing so we are able to move from one node to another then the second node is said to be reachable from the first node. The notion of traversing a network's directed graph is important when we describe information flow through a network. The concept of connectedness allows us to make this precise. We formally present this in the following definition.

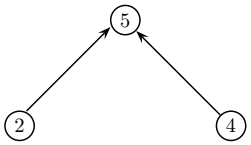
**Definition 2.4** (Connectedness)**.** A *directed path* from $x_0$ to $x_k$ is a non-empty directed graph $P_{0,k} = (V_{0,k}, E_{0,k})$ of the form $V_{0,k} = \{x_0, x_1, \ldots, x_k\}$ and $E_{0,k} = \{x_0x_1, x_1x_2, \ldots, x_{k-1}, x_k\}$, where the $x_i$ are all distinct. That is, consecutive edges in a connected path share a common endpoint; the terminal vertex of one edge is the initial vertex of the following edge. If there is a directed path $p$ from a vertex $x_i$ to a vertex $x_j$ then we say that $x_j$ is *reachable* from $x_i$ via $p$. A directed graph is called *strongly connected* if every vertex is reachable from every other.



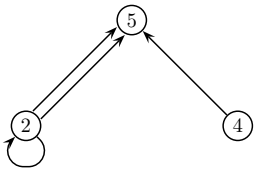

(a) A directed graph $G$.




(b) A subgraph of $G$ with the vertex set $V = \{2, 4, 5\}$.




(c) Another subgraph of $G$ with the vertex set $V = \{2, 4, 5\}$.




(d) The subgraph of $G$ induced by the vertex set $V = \{2, 4, 5\}$.

**Figure 2.3:** An example of a directed graph and three of its subgraphs.

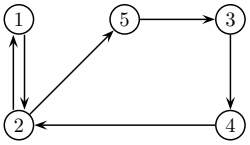



**Figure 2.4:** A strongly connected directed graph.

□

For example, let us reconsider the graph shown in Figure 2.3(a): vertex 3 is reachable from vertex 2 via a directed path $((2, 5), (5, 3))$, and vertex 2 is not reachable from vertex 3 because no directed path exists from vertex 3 to vertex 2. Furthermore, consider Figure 2.4: it shows a strongly connected directed graph.

### Subgraph Boundaries

In Chapter 6 our A-type crossover operator requires consideration of the boundary of a subgraph of a directed graph. So, we present the following definition to make our notion of a boundary precise.

**Definition 2.5** (Subgraph Boundaries)**.** Consider a directed graph $G$ with a subgraph $S$. We call an arrow that has one end vertex in $S$ and one end vertex in the complement of $S$ a *bridge* of $S$. A bridge of $S$ that has its target vertex in $S$ is called an *inbridge* of $S$. A bridge of $S$ that has its source vertex in $S$ is called an *outbridge* of $S$. An end vertex of a bridge of $S$ that is an element of $S$ is called a *proximal node*. An end vertex of a bridge of $S$ that is

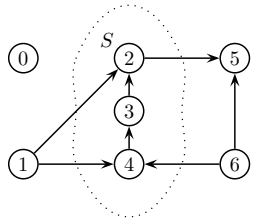



**Figure 2.5:** A directed graph $G$, with a subgraph $S$.

an element of the complement of $S$ is called a *distal node*. The set of all proximal vertices is called the *proximal boundary* of $S$. The set of all distal vertices is called the *distal boundary* of $S$.

□

For example, consider the directed graph $G = (\{0,1,2,3,4,5,6\}\ ,\ \{(1,2),\ (1,4),\ (2,5),\ (3,2),\ (4,3),\ (6,4),\ (6,5)\})$ and the subgraph $S = (\{2,3,4\},\{(3,2),(4,3)\})$. We show this graph in Figure 2.5. From this graph we can identify four bridges between $S$ and $G - S$, these are $(1,2)$, $(1,4)$, $(2,5)$, and $(6,4)$; the inbridges are $\{(1,2),(1,4),(6,4)\}$ and the only out-bridge is $\{(2,5)\}$. Furthermore we see that the proximal boundary of $S$ is $\{2,4\}$; and the distal boundary of $S$ is $\{1,5,6\}$.

### Radial Subgraph

Here we present a class of subgraph which we call a radial subgraph. In Chapter 6 we hypothesize that this is useful for our A-type crossover operator. Given a directed graph we construct a radial graph in the following way. We choose a node, which we call the centre, then collect the nodes that we encounter when we traverse non-directed paths from the centre. First, we consider paths of length one, then we consider paths of length two. We continue this procedure until we have collected a specified number of nodes. The collected nodes give a *radial set* of nodes. We call the corresponding subgraph a *radial subgraph*. We formalize in the following definition.

**Definition 2.6** (Radial Subgraph)**.** Consider a directed graph $G$. A *radial set* in $G$ about a centre $v_0$ of size $N$ is the subgraph of $G$ returned by the following algorithm.

Let $R_0$ denote a set of vertices that initially only contains $v_0$. We construct the set, $D$, of distal nodes of the subgraph induced by $R_0$. We construct a new set $R_1$ by first copying all nodes of $R_0$ into $R_1$, and second randomly choosing nodes from $D$ and moving them from $D$ to $R_1$ until either $D$ is empty or $R_1$ contains $N$ nodes. We repeat this process: we add to $D$ the distal nodes of the subgraph induced by $R_1$; then we construct a new set $R_2$ by first copying all nodes of $R_1$ into $R_2$, and second randomly choosing nodes from $D$ and moving them from $D$ to $R_2$ until either $D$ is empty or $R_2$ contains $N$ nodes. We repeat this

process until either $|R_n| = N$ or $|R_n| = |G|$. If this process terminates and $|R| < N$ then the algorithm fails because the specified radial set does not exist. If this process terminates and $|R_n| = N$ then the algorithm returns $R_n$.

The subgraph induced by $R_n$ is a *radial subgraph* in $G$ about the centre $v_0$ of size $N$.

□

For example, in Figure 2.6 we present a directed graph and three of its radial subgraphs of various sizes about a specified centre node. Figure 2.6(a) displays a directed graph. We choose vertex 4 to be a centre and we generate several radial sets about that centre, from these sets we construct the corresponding induced radial subgraphs. Three such radial subgraphs are shown.

### 2.2.1 Groups and Graphs

In Chapter 8 we apply ideas of symmetry to networks. To formulate testable hypotheses we do two things: we specify our ideas of symmetry in terms of groups, and we define how our symmetries can act on networks. The next two definitions aid our presentation in Chapter 8.

First we discuss the symmetric group $S_n$, where $n$ is some positive integer. The elements of this group are all permutations of the set of $n$ elements, and the binary operator is composition. For example, the permutation group $S_3$ has six elements, namely, {1,(12)(13),(23),(123),(1,3,2)}. These group elements reorder entries of a triple as follows:

$$\begin{aligned}
(1,2,3) &\xrightarrow{1} (1,2,3)\\
(1,2,3) &\xrightarrow{(12)} (2,1,3)\\
(1,2,3) &\xrightarrow{(13)} (3,2,1)\\
(1,2,3) &\xrightarrow{(23)} (1,3,2)\\
(1,2,3) &\xrightarrow{(123)} (2,3,1)\\
(1,2,3) &\xrightarrow{(132)} (3,1,2)
\end{aligned}$$

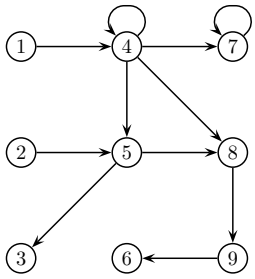
In Chapter 8 we use subgroups of $S_n$ to decrease the search space of evolutionary algorithms.

Next we define a group action (see [28, p33]).

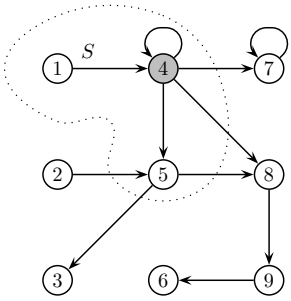
**Definition 2.7.** Let $B$ be a group and $S$ be a set. An *action* of $B$ on $S$ is a function $\rho : B \times S \to S$ such that $\rho(g, \rho(h, s)) = \rho(gh, s)$ and $\rho(1, s) = s$ for every $g, h \in B$ and every $s \in S$.

□

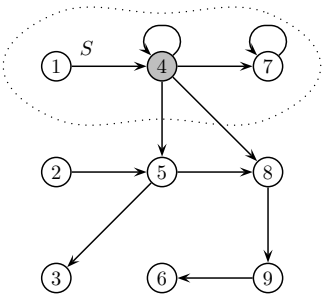
For example, let us consider the group $S_3$ acting on three dimensional Euclidean space $\mathbb{R}^3$. We can define the following group action $\rho : S_3 \times \mathbb{R}^3 \to \mathbb{R}^3$ where



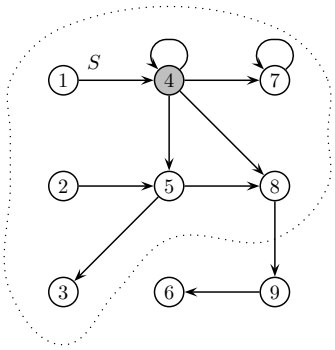

(a) A directed graph $G$.




(b) The subgraph $S$ of $G$ is one (of the six possible) radial subgraphs with the centre vertex 4 and of size three.




(c) The subgraph $S$ of $G$ is another of the radial subgraphs with the centre vertex 4 and of size three.




(d) The subgraph $S$ of $G$ is one (of the three possible) radial subgraphs with the centre vertex 4 and of size seven.

**Figure 2.6:** A directed graph and three examples of radial subgraphs in that directed graph.

$$\rho(1,(x_i,x_j,x_k)) = (x_i,x_j,x_k)$$
$$\rho((1,2),(x_i,x_j,x_k)) = (x_j,x_i,x_k)$$
$$\rho((1,3),(x_i,x_j,x_k)) = (x_k,x_j,x_i)$$
$$\rho((2,3),(x_i,x_j,x_k)) = (x_i,x_k,x_j)$$
$$\rho((1,2,3),(x_i,x_j,x_k)) = (x_j,x_k,x_i)$$
$$\rho((1,3,2),(x_i,x_j,x_k)) = (x_k,x_i,x_j)$$

for all $x_i, x_j, x_k \in \mathbb{R}$.

Next we use the definition of a group action to define a group invariant function. Our ideas for decreasing a search space, presented in Chapter 8, are only applicable to searches for group invariant functions.

**Definition 2.8.** Consider two sets $X$ an $Y$, a group $B$, and a group action $\rho$ of $B$ on $X$. A function $f : X \to Y$ is *$B$-invariant* if $f(\rho(g,x)) = f(x)$ for all $x \in X$ and $g \in B$.

□

For example, consider the norm of a triple in three dimensional Euclidean space, $f_{||} : \mathbb{R}^3 \to \mathbb{R}^3$, where $f_{||}(x_i,x_j,x_k) = \sqrt{x_i^2 + x_j^2 + x_k^2}$ for all $x_i, x_j, x_k \in \mathbb{R}$. The image of this function is independent of the order of the input variables. We now have the language to make this statement precise: $f_{||}$ is $S_3$-invariant.

Now, we formalize the notion of mapping one graph to another [29, p5].

**Definition 2.9** (Graph Morphisms)**.** Consider two directed graphs $G = (E,V)$ and $G' = (E',V')$. A *graph morphism* of $G$ is a function $\phi : G \to G'$ where $\phi$ is a pair $(\phi_E, \phi_V)$ consisting of an edge function $\phi_E : E \to E'$ and a vertex function $\phi_V : V \to V'$, such that for every edge $(v_1,v_2) \in E$ the condition $\phi_E(v_1,v_2) = (\phi_V(v_1), \phi_V(v_2))$ holds. If both $\phi_E$ and $\phi_V$ are bijections then $\phi$ is called a *graph isomorphism*. If $\phi$ is an isomorphism and $G = G'$ then $\phi$ is called a *graph automorphism*. We say that a vertex $v$ is *fixed* under $\phi$ if $\phi_V(v) = v$. Similarly, we say that an edge $e$ is *fixed* under $\phi$ if $\phi_E(e) = e$.

□

For example, consider the graph morphism illustrated in Figure 2.7 and the graph automorphism illustrated in Figure 2.8.

Second, we formalize the notion of imposing the properties of a group on a graph. In Definition 2.2.1 we constructed a function $\rho$ to formalize the notion of a group acting on a set. However, to clarify the notion of a group acting on a graph we express group actions in terms of automorphisms. First let us articulate our claim that group actions are equivalent to automorphisms.

**Claim 2.1** (Group acting on a Graph)**.** Consider a set $X$, a group $B$, and the group $\mathrm{Sym}(X)$ whose elements are bijections $f : X \to X$ and whose binary operation is function composition.


**Figure 2.7:** Two directed graphs $G$ and $G'$, and a graph morphism $f : G \to G'$ where $f_V(v_1) = f_V(v_2) = v'_1$, $f_V(v_3) = v'_2$, $f_V(v_4) = v'_3$, and $f_E(e_1) = f_E(e_2) = e'_1$, $f_E(e_3) = e'_2$.


**Figure 2.8:** A directed graph $G$ and a graph automorphism $f : G \to G$ where $f_V(v_1) = v_2$, $f_V(v_2) = v_3$, $f_V(v_3) = v_1$, and $f_E(e_1) = e_2$, $f_E(e_2) = e_3$, $f_E(e_3) = e_1$.

For any action $\rho$ of $B$ on $X$ we can construct a homomorphism $R : B \to \mathrm{Sym}(X)$ by $R(g) = \rho(g, x)$ for some $x \in X$. Conversely, for any homomorphism $R : B \to \mathrm{Sym}(X)$ we can define an action $\rho$ of $B$ on $X$ by $\rho(g, x) = R(g)(x)$, where $g \in B$ and $x \in X$.

□

For further details see [28, p34]. Mindful of this equivalence between actions of $B$ on $X$ and homomorphisms from $B$ to $\mathrm{Sym}(X)$ we now present a group acting on a graph in terms of graph automorphisms. For the following definition we reproduce material from Gross and Tucker [29, p21].

**Definition 2.10** (Group acting on a Graph)**.** Let $G$ be a graph and let $B$ be a group and suppose for each element $b \in B$, we are given a graph automorphism $\phi : G \to G$ and such that the following two conditions hold: if 1 is the group identity then $\phi_1 : G \to G$ is the identity automorphism; and, for all $b, c \in B$, $\phi_b \circ \phi_c = \phi_{bc}$. Then the group $B$ is said to *act on the graph* $G$.

□

## 2.3 Bitstrings

In this section we introduce language to describe strings of bits. This is useful because our research uses Boolean artificial neural networks that accept sequence of bits.

| $A$ | $B$ | $A \oplus B$ |
|---|---|---|
| 1 | 1 | 0 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 0 | 0 | 0 |

**Table 2.1:** A truth table defining the Boolean operation Exclusive-OR.

### Data Packets

In latter chapters we need to be able to enter several strings of bits into a network, and also collect several such strings as output. To clarify this process we define a data packet as a list of strings of bits.

**Definition 2.11** (Data Packet)**.** A *data packet* is an $m \times n$ matrix each entry of which is an element of $\mathbb{Z}_2$. We use the symbol $M[m, n]$ to denote the set of all $m \times n$ data packets.

□

### Comparing Data Packets

Also we need to compare bitstrings, so we introduce the Hamming distance. The Hamming distance between two bit strings is the number of bit positions in which the two strings differ [30, ch. 13]. This is useful when we compare expected and actual output data packets. We define the Hamming distance in terms of data packets below.

**Definition 2.12** (Hamming Distance)**.** Consider two $m \times n$ data packets $P = [p_{ij}]$, $Q = [q_{ij}]$. The *Hamming distance* between $P$ and $Q$, which we denote by $H(P, Q)$, is the number of pairs $(i, j)$ such that $p_{ij} \neq q_{ij}$. The *normalised Hamming distance* between $P$ and $Q$, which we denote by $\hat{H}(P, Q)$, is $\frac{1}{(m \times n)} \times H(P, Q)$.

□

### Boolean Functions

A *Boolean function* is any function $f : (\mathbb{Z}_2)^m \to (\mathbb{Z}_2)^n$. Note that some authors define a Boolean function as the special case where $n = 1$ (for example Siu *et al.* [31, p] distinguish between Boolean functions and multi-output Boolean functions). Often Boolean functions are defined by *truth-tables* and these tables are used to investigate the equivalence of two Boolean expressions. For example, in Table 2.1 we define Exclusive-OR.

If we only consider single column data packets then Boolean functions can be defined in terms of sets of input-output pairs of data packets. In Figure 2.9 we illustrate such a set for Exclusive-OR.

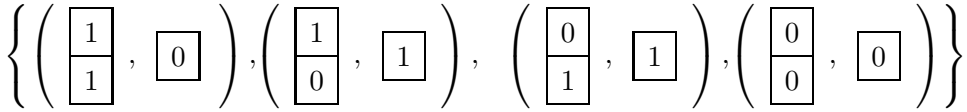


**Figure 2.9:** Expressing the Boolean function Exclusive-OR using single column data packets.

We introduced data packets so that we can precisely describe input and output from an A-type. In general, the input and output from our A-types is sequential. So, our A-types employ data packets that have more than one column. This motivates us to extend the notion of a Boolean function to a function with input-output pairs of data packets of arbitrary length.

**Definition 2.13** (Clamped Boolean functions)**.** Consider a Boolean function $f : \mathbb{Z}^m \rightarrow \mathbb{Z}^n$. Consider the set $X = \{x | x \in M[l, m]$ and all columns of $x$ are identical $\}$. Also consider the set $Y = \{y | y \in M[l, n]$ and all columns of $y$ are identical $\}$. Furthermore, consider the function $g : X \rightarrow Y$ such that for every $x \in X$ the data packet in the $i$th column of $g(x)$ equals $f($ the $i$th column of $x$ ) where $i$ indexes all columns in $x$. We call $g$ the *clamped* case of the Boolean function $f$.

□

We also introduce the following more general extension of a Boolean function.

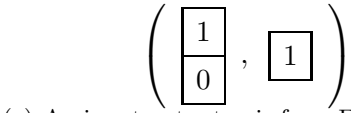
**Definition 2.14** (Columnwise Boolean functions)**.** Consider a Boolean function $f : \mathbb{Z}^m \rightarrow \mathbb{Z}^n$. Consider the set $X = \{x | x \in M[l, m]\}$. Also consider the set $Y = \{y | y \in M[l, n]\}$. Furthermore, consider the function $g : X \rightarrow Y$ such that for every $x \in X$ the data packet in the $i$th column of $g(x)$ equals $f($ the $i$th column of $x$ ) where $i$ indexes all columns in $x$. We call $g$ the *columnwise* case of the Boolean function $f$.

□

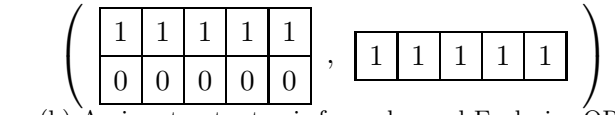
For example, consider the data packets shown in Figure 2.10.

## 2.4 Presentation of Algorithms

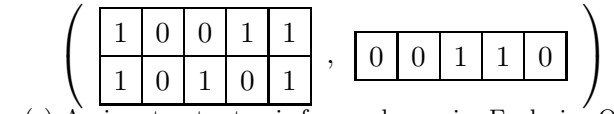
We use two formats to present algorithms. One gives a brief overview and we call this an *outline*; the other gives greater detail and we call this a *description*. For example, consider the task of having breakfast, we present this as the algorithm *have_breakfast*. We outline this algorithm in Table 2.2 and give greater detail in Table 2.3. If an outline of an algorithm is sufficient then we do not present a description of that algorithm. If our presentation of the algorithm requires more detail then we use both an outline and a description in conjunction.




(a) An input-output pair from Exclusive-OR.




(b) An input-output pair from clamped Exclusive-OR.




(c) An input-output pair from columnwise Exclusive-OR.

**Figure 2.10:** A pair from columnwise Exclusive-OR of length five.

| *have_breakfast* |
|---|

If we don't have enough time then we have a banana, otherwise we perform the following steps. Repeat five times.

1. Play favourite song.
2. Prepare two wheat biscuits.
3. Eat wheat biscuits.

**Table 2.2:** A outline of *have_breakfast*.

**have_breakfast**$(t, t_{min})$ is an algorithm describing how the author has breakfast. An outline of this algorithm is given in Table 2.2.

| Parameters for the Initial Population | | |
|---|---|---|
| *Type* | *Parameter* | *Description* |
| $\mathbb{Z}^+$ | $t$ | Time (in seconds) available. |
| $\mathbb{Z}^+$ | $t_{min}$ | Minimum time required to eat ten wheat biscuits. |

The algorithm has the following steps

if$(t < t_{min})$

*then*

Eat a banana.

*otherwise*

Repeat five times.

1. *Choose song:* Choose and play a music track to listen to. Preferably an uplifting song.
2. *Prepare:*
   a) Put two wheat biscuits into a bowl.
   b) Put milk into bowl so that the biscuits are half submerged.
   c) Evenly distribute about a dessert spoonful of sugar onto the biscuits.
3. *Eat:* Consume both wheat biscuits.

**Table 2.3:** A description of *have_breakfast*.

# 3 Machine Learning

## 3.1 Aims of this Chapter

The first aim of this chapter is to give a very brief and biased overview of machine learning (ML). We focus on evolutionary algorithms, artificial neural networks, and Alan Turing's pioneering work in these fields. This provides the context in which our research was performed.

The second aim of this chapter is to introduce Alan Turing's pioneering work on ML. Turing discovered ideas that are active fields of modern ML research.

## 3.2 Machine Learning and Artificial Intelligence

Machine learning is the field of study of algorithms that improve computer programs with its experience. Often ML is classified as a subset of artificial intelligence (AI). When we consult the Oxford English Dictionary to clarify this we get the following two definitions.

**artificial intelligence, n.** The capacity of computers or other machines to exhibit or simulate intelligent behaviour; the field of study concerned with this. Abbreviated AI.

**machine learning, n.** *Computing* the capacity of a computer to learn from experience, i.e. to modify its processing on the basis of newly acquired information.

Artificial intelligence is a vast subject, it has a long history, and it has been influenced by many disciplines [32]. In the second half of the twentieth century electronic computers began their rapid advance and some AI researchers heralded great predictions. The following quote from Marvin Minsky [33, p2] is evidence of the optimism of the time.

> within a generation, I am convinced, few compartments of intellect will remain outside the machine's realm—the problem of creating "artificial intelligence" will be substantially solved.

Since then such optimism has proven to be premature, at best *. Many disciplines investigate AI including philosophy, psychology, neuroscience, and computer science. Often the research methods from separate disciplines are substantially different.

Some AI researchers attempt to model the biology of organisms that exhibit intelligence†.

*Copeland[34] provides a good introduction to AI; also the first episode of the BBC documentary *Visions of the Future* [35] provides an entertaining and informative description of recent developments of AI.

†One of the most elaborate of such projects is the 'Blue Brain Project' [36], which aims to accurately simulate the human brain.

This approach potentially offers insight into the behaviour of biological systems; for instance, the mammalian brain and groups of social animals. Inspiration for this approach is obvious, but it is not obvious whether it will provide a means of constructing a machine capable of human-level intelligence. Some AI researchers employ methods that have a rather tenuous connection with biology. For example, kernel methods map data to abstract mathematical spaces to aid the discovery of rules that govern that data [11]. Steels [37] gives the analogy that we don't require aircraft designers to make machines that have flapping wings or feathers, and it isn't necessary that AI research should attempt to mimic biology. Many approaches to AI are inspired by biology [38] but often the biological analogies are only loosely adhered to. For example in most artificial neural networks the neurons and their interconnections are extremely simple in comparison to any seen in biology. Another example is multiparent recombination in evolutionary computing. In evolutionary computing a 'child' algorithm may be the consequence of more than two 'parent' algorithms [39].

As a subset of the subject, ML may make useful contributions to AI. But, separate from the immense goals of AI, ML already provides algorithms that are in use today. For example such algorithms are used in facial pattern recognition, stock market prediction, and the analysis medical samples [40].

## 3.3 A Learning Discussion

In this section we touch upon the theory of learning. Our current presentation simply serves to provide appropriate language for following sections. Herbrich [11, ch 1] provides a clear introduction to the classification of learning. Herbrich defines learning as the task of finding a general rule that explains data given only a sample of limited size [11, p1]. Our current presentation simply serves to provide appropriate language for following sections.

Now we present three classes of leaning, namely supervised learning, unsupervised learning and reinforcement learning. First we consider supervised learning (also called concept learning [40, p20]). This is learning with a set of examples. Duda *et al.* [41, p16] introduce supervised learning as: "a teacher provides a category label or cost for each pattern in the training set, and seeks to reduce the cost of these patterns." For instance, given a set of $(x_i, y_i)$ pairs we may seek to discover a polynomial that fits that set. Using supervised learning the performance of a candidate solution $f$ is assessed by examining the difference between $y_i$ and $f(x_i)$ for all $(x_i, y_i)$ pairs. Second, we consider unsupervised learning. Duda *et al.* [41, p17] introduce this with the following sentence: "In unsupervised learning or clustering there is no explicit teacher, and the system forms clusters or 'natural groupings' of the input patterns." One example of unsupervised learning is Hebbian learning in neural networks [42, p368]. Loosely stated, Hebbian learning is the strengthening of connections between neurons that fire simultaneously. This requires only input, rather than input-output pairs. Third, we consider reinforcement learning. Duda *et al.* [41, p17] summarise this as: "analogous to a

critic who merely states that something is right or wrong, but does not specifically say how it is wrong". Unlike supervised learning, for reinforcement learning input-output pairs are not available, but the performance of output can be assessed.

The learning tasks that we perform with Turing A-type networks are best described as supervised learning[‡]. To aid our discussion we employ the following definition.

**Definition 3.1** (Supervised Learning)**.** Consider two sets $X$ and $Y$ and a function $c : X \to Y$. Also consider two sets $T \subseteq X$ and $Z \subseteq Y$ and a function $c_T : T \to Z$, where $c(x) = c_T(x)$ for all $x \in T$. *Supervised Learning* is the search for a function $s : X \to Y$ that approximates $c$, to some prescribed accuracy, using $c_T$. We say that $s$ is a *solution* and $c$ is the *concept* being learnt. We call the set of all possible functions in the search the *hypothesis space* of the search. We call each function that the search encounters in the hypothesis space a *candidate solution*. Furthermore, we call the set $\{(x, c_T(x))|x \in T\}$ the *training set* and each element of this set a *training example*.

□

For example, if we employ supervised learning to searching for the concept of addition $+ : \mathbb{R}^2 \to \mathbb{R}$ we may use the set $\{((1,2),3),((1,3),4),((4,6),10),((7,7),14)\}$ as a training set.

## 3.4 Evolutionary Computation

Evolutionary computing is the application of ideas from evolutionary biology to machine learning. An evolutionary algorithm (EA) is an algorithm used for this application. An EA is a form of beam search—a supervised learning strategy that was developed independent of EAs [43, p27]. Evolutionary computing is an active field of research and has wide technological application: from engineering problems [44] to stock market prediction [45]. Mitchell [40, p250] lists the following three items as motivation for the use of EAs.

- Evolution is known to be a successful robust method for adaptation within biological systems.
- [EAs] can search spaces of hypotheses containing complex interacting parts, where the impact of each part on overall hypothesis fitness may be difficult to model.
- [EAs] are easily parallelized and can take advantage of the decreasing costs of powerful computer hardware.

[‡]Note that when we train our A-type networks it may be more appropriate to discuss this in terms of reinforcement learning because learning is not solely dependent on the training set: we also consider the size of the candidate solutions.

Evolutionary Computing is over fifty years old. In 1948 Turing [1] suggested a "genetical search" and in the late 1950s research was published in many fields that may be categorised as evolutionary computing[§]. For a more detailed introduction to Evolutionary Computing we recommend Mitchell [40, chapter 9], Eiben *et al.* [47] and Banzhaf *et al.* [43]. We borrow heavily from these sources in the following presentation.

### 3.4.1 Employing the Idea of Evolution

Most instances of EAs use supervised learning but they may also use reinforcement learning or unsupervised learning. [43, p29]. Here we present a prototypical EA that uses supervised learning. That is, we present an EA that searches for a function that 'fits' a set of training examples. Figure 3.1 (reproduced from [47, p17]) gives a flow chart for this algorithm and the algorithm has the following steps.

1. Randomly generate a specified number of candidate solutions; we call the multiset of candidate solutions the *population*[¶]. For each member $x$ of the population we assign a measure of how well it fits the training data. We call this measure the *fitness* of $x$.

2. Select a number of pairs of candidate solutions as parents. The fitter a candidate solution, the greater the probability that it is chosen as a parent. We call this step *parent selection*.

3. For each parent pair we combine information from both parents to produce new candidate solutions. These new candidate solutions are added to the population. We call this step *recombination*.

4. Randomly select members of the population. For each selected member we copy it and slightly modify the copy. Each modified copy is added to the population. We call this step *mutation*.

5. Select several members of the population and delete them. We call this step *survivor selection*.

6. If the population contains a solution then return a solution and terminate the search. Otherwise, steps 2-5 above are repeated. We maintain a variable called generation. Throughout the first iteration of the above steps the generation is assigned zero, at the beginning of each subsequent iteration the generation is incremented by one. If the generation exceeds a specified value then the search is terminated without returning a solution.

[§]Fogel *et al.* [46] present an account of pioneering EC research and this account includes reprints of early publications.

[¶]The population is a *multi*set because the same candidate solution may appear more than once in the population.

The above algorithm is analogous to biological evolution. Consequently, we often refer to candidate solutions as individuals. Mindful of this biological analogy, we proceed to elaborate on aspects of EAs.

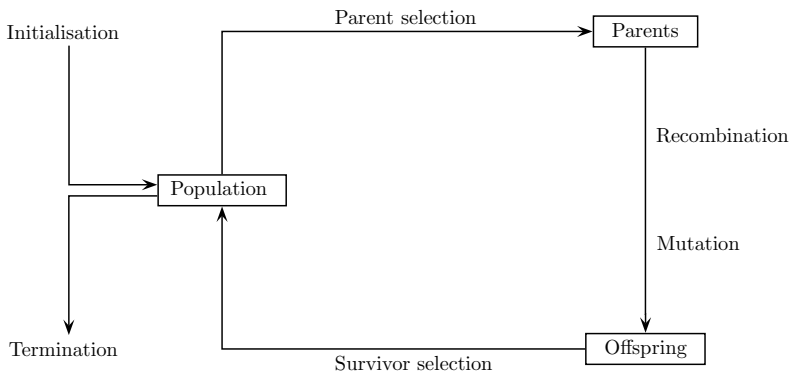



**Figure 3.1:** A flow chart illustrating our prototypical EA (reproduced from Eiben *et al.* [47, p17]).

### Evaluation of Fitness

When we calculate the fitness of a given candidate solution, we can use criteria that are independent of the training data. For example, our candidate solutions may be polynomials and we may require a solution to closely fit the training examples and be small—that is be polynomials with few terms. In this case we can calculate a candidate solution's fitness as some combination of how well it fits the training data and its size. Combining a candidate solution's size into our measure of fitness should create an 'evolutionary pressure' on the search to produce small solutions that fit the data well.

The evaluation of a candidate solution's fitness is described in terms of a fitness function. For our research, we adopt the convention of normalised fitness [43, p127] where the measure of fitness is a real number in the interval $[0, 1]$. The fitter the candidate solution, the lower the fitness—the best possible fitness equals zero. So our fitness function is a map from the hypothesis space (the space of possible candidate solutions) to $[0, 1]$.

### Selection Rules

In general, an EA has a parent selection rule, a mutant selection rule, and a survivor selection rule. These rules can be fitness-based: for a fitness-based selection rule the probability that a member of the population $x$ is selected is a function of $x$'s fitness relative to the fitnesses of all other members of the population. For example, we can perform a ranking selection [47, p60] where we order the individuals by fitness and randomly select from that order in such a

way that the higher an individual appears in that order the greater the probability that the individual is selected.

In biological evolution both parent selection and survivor selection are fitness-based, and mutant selection is not. Biological parent selection can be due to sexual selection: individuals select one another based on their assessment of others fitnesses. A striking example of sexual selection is peahens' preference for peacocks with large tail feathers (in spite of the disadvantage that these feathers incur if peacock must escape a predator). Biological parent selection can also be due to selective breeding imposed by an external agent. For example, humans select for crops and livestock that fit their needs. In biological evolution mutant selection is independent of fitness. In biological evolution survivor selection is fitness based. The obvious example of this is natural selection. By making the survivor selection rule fitness-based we can implement an analog to natural selection. Furthermore, we can implement ideas like aging by making the probability of an individual $x$ being selected as a survivor decrease as a function of the number of generations since $x$ joined the population.

### Variation Operators

We call mutation and recombination *variation operators*. Consider an EA with a hypothesis space $H$. Mutation is a function $m : H \to H$. Recombination is a function $r : [H]^n \to H$, where $n \in \mathbb{Z}$ and $n > 1$. If $n = 2$ then $r$ is called *crossover*. If $n > 2$ then $r$ is called *multiparent recombination*. Note that multiparent recombination is not seen in nature; so, computer simulation of EAs offers a means of investigating why there is no multiparent recombination in nature.

### Sex is Useful

Evolution does not require recombination, but most EAs incorporate a recombination operator. The motivation for this is two-fold. First, recombination is useful in biology. The evolution of most multicellular organisms involves the exchange of genetic material via sex. It is likely that this process is beneficial in spite of the burden associated with it. For example, often a great deal of time and energy is expended in courtship, yet sexual reproduction is common. Second, there are strong information theoretic arguments in favour of using recombination operators. Mackay [30, ch.19] presents a simple model of the information gained during evolution and demonstrates that sexual recombination is advantageous. In summary of this presentation Mackay writes.

> These results quantify the well know argument for why species reproduce by sex with recombination, namely that recombination allows useful mutations to spread more rapidly through the species and allows deleterious mutations to be more rapidly cleared from the population.

It is not clear whether recombination operators are beneficial in EAs. Although EAs are inspired by biology links between the two fields are often tenuous. In most cases biological chromosomes are immensely more complex than that of an EA. Similarly, variation operators in biology are usually incredibly sophisticated in comparison to those in an EA. Whether recombination operators are useful in EAs is currently an active area of research [5] [6] [7]

### 3.4.2 Implementing Evolutionary Algorithms

When we defined learning we kept it rather general. When implementing a learning method we have to be precise about how we encode possible candidate solutions. EAs are no exception to this. In biological evolution the variation operators act on chromosomes. In an EA we need to define the 'chromosomes' of candidate solutions so that we can precisely define the the EA's variation operators. The *chromosome* of an EA is the data structure that we employ to encode each candidate solution. Many authors refer to these data structures as an EA's 'representation' [43] [47] [40], but we use the term chromosome for fear of overusing the former term[‖].

Next we present three classes of EA chromosomes [47, p112].

#### Linear Chromosomes

In 1975 Holland [48][**] presented evolutionary computing in terms of linear chromosomes. A linear chromosome is a string of characters that encodes a candidate solution. Consider the following concrete example. Consider a small tennis club that decides to organize a six week schedule for Sunday morning games. The club only has one court available, and only time for two games on each Sunday morning. There are five teams participating. We have to schedule which teams play one another and when they do so. We encode this problem so that it can be solved with an EA. First let us encode the permutations of pairs of teams as shown in Figure 3.2(a). So, for example, if there is a game between team $B$ and team $D$ then we encode it with 5. Our encoding of a six week schedule is illustrated in Figure 3.2(b). Each box in the diagram can accept a decimal that encodes a game; there are twelve boxes and they are ordered according to the chronology of the games. We call this encoding a linear chromosome. For example, Figure 3.2(c) shows one possible schedule. In this schedule on the first Sunday the first game is between team $C$ and team $E$, and the second game is between team $A$ and $B$; also, on the second Sunday the first game is between team $B$ and team $C$.

Given a fitness function and variation operators we can use an EA to search for satisfactory tennis schedule. First, we discuss a fitness function. If we examine the schedule in

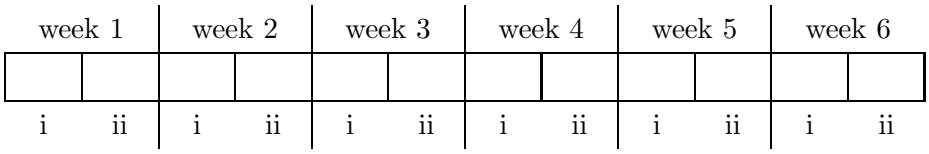
[‖] Representation has a specific meaning in group theory and in Chapter 5 we define how an A-type can represent a function.

[**] In this book Holland introduced the term *genetic algorithm*. Often this term is used interchangeably with the term *evolutionary algorithm*.

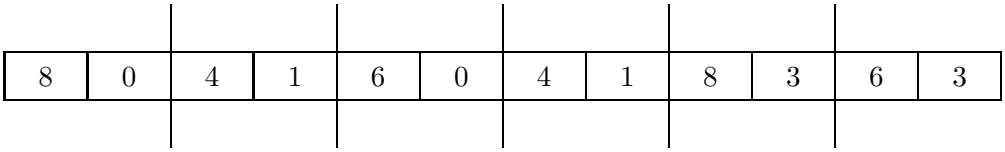
Figure 3.2(c) we discover that team $D$ never plays a game. This seems like a rather unsatisfactory schedule. One criteria for a satisfactory schedule may be that all teams play a similar number of games. There may be other criteria, for example playing the first game of a Sunday may be preferable to playing the second game—if this is the case then the members of team $A$ may be displeased because they never play in the first game of a Sunday. If the criteria for a satisfactory schedule can be quantified then we can devise a fitness function; that is, a function that maps any schedule to the interval $[0, 1]$. Second, let us describe two variation operators that can act on our schedules. We can mutate a schedule $S$ by randomly selecting a digit in $S$ and adding 1 (modulo 10) to it, this is illustrated in Figure 3.3. We can recombine two schedules by transposing sections, this is illustrated in Figure 3.4.

| team | team | encoding |
|---|---|---|
| A | B | 0 |
| A | C | 1 |
| A | D | 2 |
| A | E | 3 |
| B | C | 4 |
| B | D | 5 |
| B | E | 6 |
| C | D | 7 |
| C | E | 8 |
| D | E | 9 |

(a) Encoding competitions. For instance, 7 encodes a game between team $C$ and team $D$.




(b) A chromosome that can store a candidate schedule.




(c) A candidate schedule.

**Figure 3.2:** Encoding a scheduling problem with linear chromosomes.

The above example demonstrates an important property of linear chromosomes, namely both mutation and recombination are easily implemented.

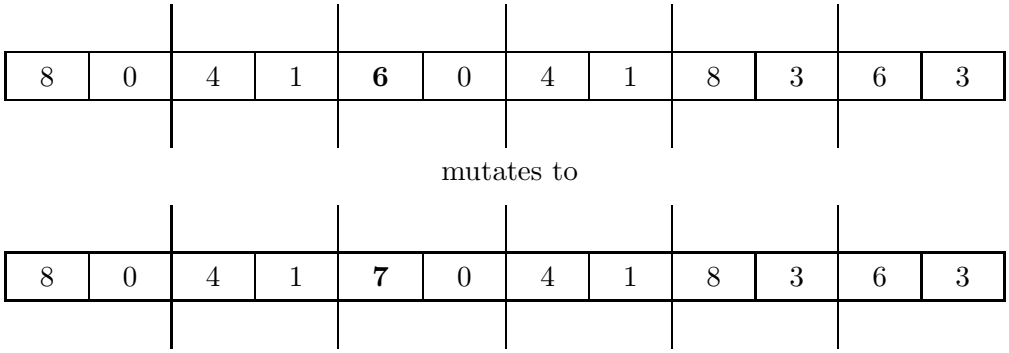



**Figure 3.3:** A simple mutation: one of the digits in a schedule is incremented by one to give a mutant schedule.

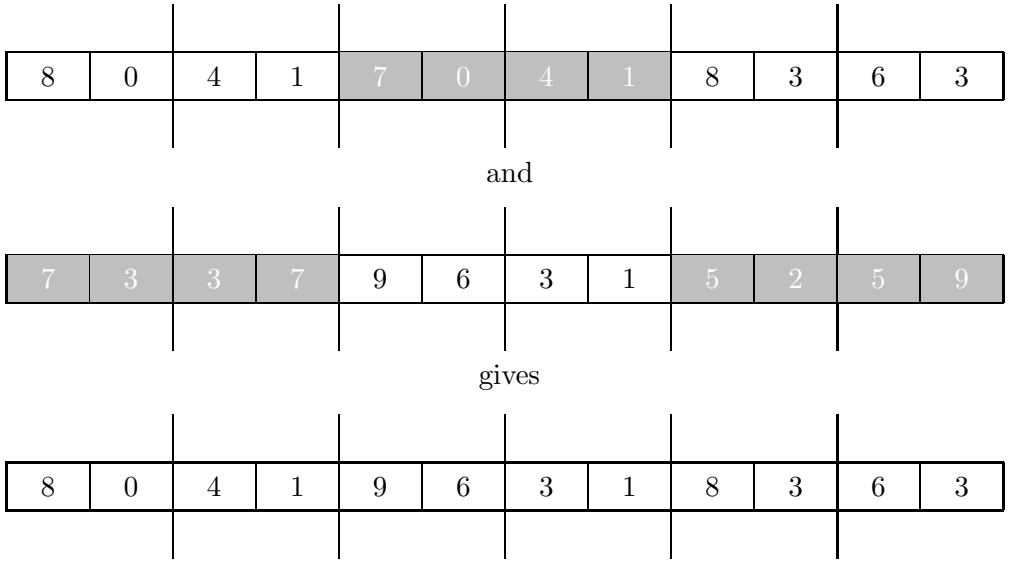
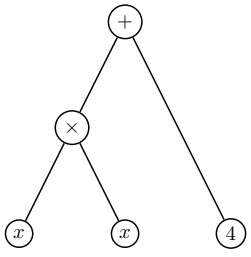



**Figure 3.4:** A simple recombination: sections (shaded) of two schedules are deleted and the remaining sections (non-shaded) are combined to give a new schedule.

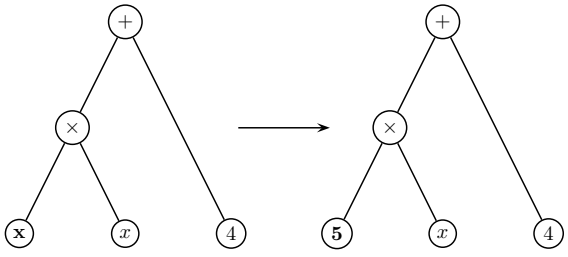
### Tree Chromosomes

There are problems that when tackled with an EA are best represented in a tree data structure. For instance, searching for a polynomial that fits a particular data set. In such a search polynomials can be represented as trees. For example, Figure 3.5(a) shows that we can represent polynomials with trees, where the nodes represent binary operators and the leaf nodes represent either variables or real numbers. With these tree chromosomes we can devise mutation and recombination operators, as shown in Figure 3.5(b) and Figure 3.5(c). The tree shown in Figure 3.6(b) represents the computer program given in Figure 3.6(a). Like the trees shown in Figure 3.5, these program trees can be mutated and recombined. Consequently, this provides a scheme for evolving computer programs[††].



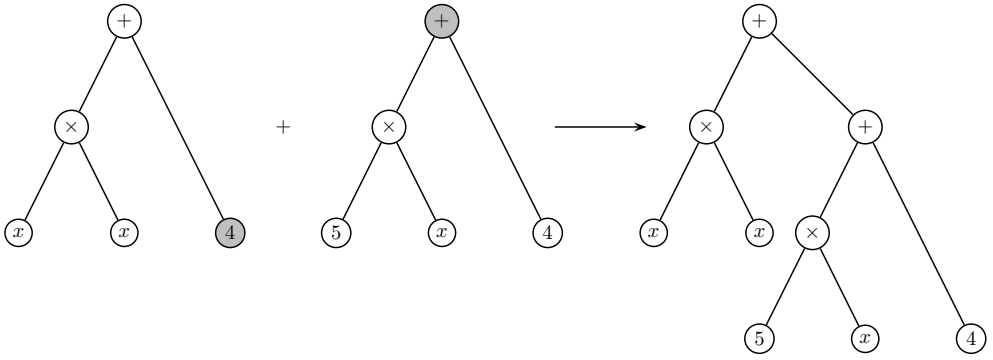

$x^2 + 4$

(a) A tree chromosome that represents a simple polynomial.




$x^2 + 4$ $\qquad$ $5x + 4$

(b) A simple mutation.




$x^2 + 4$ $\qquad$ $5x + 4$ $\qquad$ $x^2 + 5x + 4$

(c) A crossover: grafting subtrees together.

**Figure 3.5:** Tree chromosomes

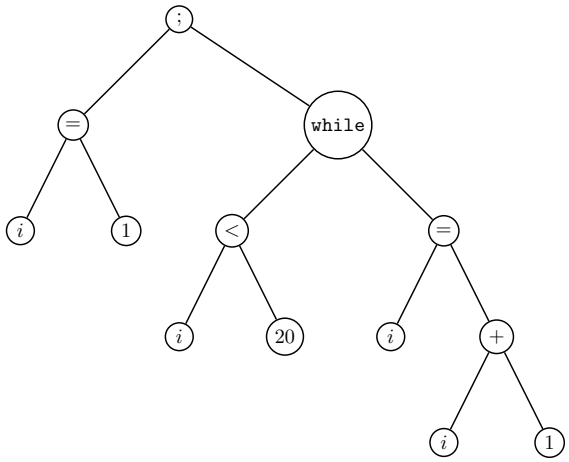
### Graph Chromosomes

Graphs are another data structure that can be used as chromosomes in EAs. When Turing suggested a 'genetical search' the chromosomes were graphs of NAND gates: EAs may have

[††]Koza [49] introduced the term *genetic programming* to describe this evolution of computer programs.

```
i = 1;
while(i < 20)
{
i = i + 1;
}
```

(a) A computer program.




(b) A parse tree for the program.

**Figure 3.6:** Representing a computer program with a tree chromosome (reproduced from [47, p104]).

first been proposed, by Turing, with graph chromosomes. In the late 1960's Fogel *et al.* researched EAs using finite state machines [50], and in more recent times Teller and Veloso used EAs with register machine chromosomes [51], both employ graph chromosomes. The use of EAs to train artificial neural networks (see the next section) is an active area of research, and often this too requires graph chromosomes. The current section is rather brief because much of the remainder of this thesis details evolution of a particular class of graph chromosomes.

### Evolutionary Strategies

Evolutionary algorithm encompasses a broad class of algorithms. Even if we specify the class of chromosome that is used there is much freedom when specifying the algorithm. For example, the population may be partitioned with a low probability of a member of one partition migrating to another partition. These partitions are referred to as *islands of evolution* [47, p158]. Another example is the freedom in selection. An alternative to ranking selection is tournament selection: randomly choose two individuals and the fittest survives [47, p63]. If we tackle a given problem with an EA then we must specify the particulars of the algorithm. All such algorithms are 'evolutionary' because candidate solutions contain hereditary information from previous generations.

### Complementing Biology

Biological evolution inspired evolutionary computing, and the latter helps us investigate the former. Darwin and Wallace's momentous discovery of natural selection was made from complex real-world data. Biology is a study of very complex systems and often evolution of

these systems is only discernible after a considerable number of generations. Evolutionary computing provides a controlled environment to test evolutionary ideas [52]. For example we can investigate why multiparent recombination is not seen in biology.

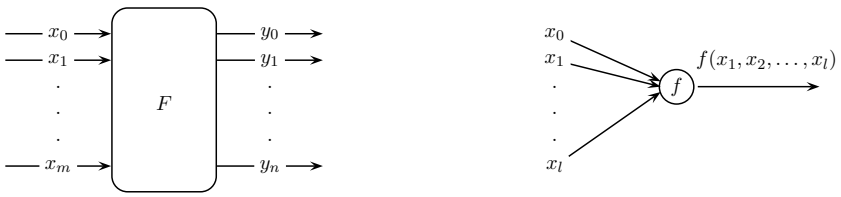
## 3.5 Artificial Neural Networks

In 1943 McCulloch and Pitts published a seminal paper [53] that sparked Artificial Neural Networks (ANNs). Today ANNs have wide application—they even track villains [54] and drive fast cars [55]. Although inspired by the mammalian brain, most implementations of ANNs are relatively simple, yet they provide exciting results. Haykin [42, p9] articulates this point with the following statement:

> The artificial neurons that we use to build our neural networks are truly primitive in comparison with those found in the brain. The neural networks that we are presently able to design are just as primitive compared with the local circuits and the interregional circuits in the brain. What is really satisfying, however, is the remarkable progress that we have made on so many fronts. With neurobiological analogy as the source of inspiration, and the wealth of theoretical and computational tools that we are bringing together, it is certain that our understanding of artificial neural networks and their applications will continue to grow in depth as well as breadth, year after year.

In this section we provide an overview of ANNs. There are numerous implementations of ANNs, taxonomies of which are available in the literature [56] [57]. Our overview introduces the reader to ANNs, allows the reader to classify—in the context of today's terminology—networks that Alan Turing proposed (we introduce these in Section 3.7.1), and provides language required to explain our implementation of Turing's networks.

### 3.5.1 Static Networks




(a) Schematic of an ANN.

(b) Schematic of a neuron.

**Figure 3.7:** Illustrating an ANN.

An ANN is a device that takes a set of input values and returns a set of output values, we illustrate this in Figure 3.7(a). That is, an ANN maps an input vector $\mathbf{x} = (x_1, x_2, \ldots, x_m)$ to an output vector $\mathbf{y} = (y_1, y_2, \ldots, y_n)$, where (in general) $\mathbf{x} \in \mathbb{R}^m$ for a some positive integer $m$, and $\mathbf{y} \in \mathbb{R}^n$ for a some positive integer $n$. This device consists of basic devices which we call neurons. We illustrate a neuron in Figure 3.7(b). The output value of a neuron is given by the map $f : \mathbb{R}^l \to \mathbb{R}$, for some positive integer $l$. The inputs and outputs of an ANN are connected via neurons. For example consider the ANN illustrated in Figure 3.8. Let us call this network $A_a$, and let $F_a : \mathbb{R}^3 \to \mathbb{R}^2$ denote the function whose input-output pairs can be collected from $A_a$. We illustrate the connections between neurons with arrows. The input arguments of a neuron are prescribed by the arrows entering a neuron: each argument is either the value of some neuron's function or an element of the $F_a$'s domain. That is, the arrows prescribe how the neurons' functions are composed for the $A_a$'s output. Each output $y_i$ is given by a composition of neuron functions. Consider the directed graph $G_a$ whose vertices are the set $A_a$'s neurons and whose arrows are the connections between $A_a$'s neurons. If $G_a$ has no closed paths then $F_a$ can be determined by composing the functions of $A_a$'s neurons [58, p30]. In general, if the directed graph that represents an ANN's neurons and their connections has no directed loops then the ANN is called a feedforward network; otherwise, it is called a feedback, or recurrent, network.

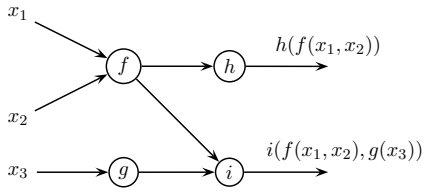



**Figure 3.8:** A simple example of an ANN, which we call $A_a$. Note that the arrows indicate how the neurons' functions can be composed to give the ANN's output. Also, note that the value associated with each arrow exiting the top left neuron is the same, namely $f(X_1, X_2)$.

### 3.5.2 Dynamic Networks

In the above section we could assume that the evaluation of each neuron's function was instantaneous. Because the networks were feedforward it was unnecessary to synchronize the output of the neurons. Time is a necessary argument for a general description of ANNs. If an ANN can be well defined without a time argument then we call it a static network; otherwise, we call it a dynamic network. Mammalian networks process with a continuous time parameter[‡‡] and some ANNs also require a continuous time parameter [60].

Often discrete time is adequate for the description of an ANN. For instance, McCulloch

[‡‡]The timing of mammalian brains is far from understood. For example, even in the absence of any theoretical physics consideration of the nature of time, research suggests that the 'clock speed' of the human brain can increase when a person is stressed [59].

and Pitts networks use discrete time [33, chap 3]. Consider the ANN shown in Figure 3.9, which we call $A_b$. This is an example of a Boolean network because each of $A_b$'s inputs and each of its neurons' inputs is an element of $\mathbb{Z}_2$. Let us examine $A_b$'s response to two inputs. Although we have specified a Boolean function for each of $A_b$'s neurons, we also must specify the initial conditions, namely the output of each neuron at time $t = 0$. So for each neuron function we choose the output at time $t = 0$ to be zero. The response of $A_b$ to the input $(0, 1)$ is presented in Table 3.1. Note that we are synchronously updating $A_b$; that is, there is an external clock and at each moment of that clock every neuron evaluates its output function[§§]. Table 3.1 shows that, although the input $(0, 1)$ is constant through all time, the output at time $t = 0$ differs to the output at later times. In this case we can attribute this instability of the initial output to our arbitrary choice of initial conditions. It is tempting (but wrong) to conclude that if we consider the output of $A_b$ after one time moment then $A_b$'s mapping is well defined, for instance $F_{A_b}((0, 1)) = 1$. Now let us consider $A_b$'s response to the input $(1, 0)$: this is shown in Table 3.2. We see that for the input $(1, 0)$ the output of $A_b$ is periodic (with a period of four). Such complex behaviour from a relatively simple network is useful; Boolean networks offer a means of investigating non-linear phenomena *in-silico* [62, p195]. For instance, Boolean networks with discrete time are used to investigate biological systems, such as gene regulatory networks [63].

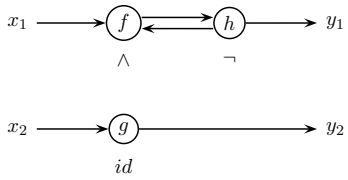



**Figure 3.9:** An ANN that maps $[\mathbb{Z}_2]^2$ to $[\mathbb{Z}_2]^2$. The neuron function $f$ is logical AND, the neuron function $g$ is the identity $id : \mathbb{Z}_2 \to \mathbb{Z}_2$, and the neuron function $h$ is negation.

### 3.5.3 Dynamically Driven Networks

Because dynamic ANNs can exhibit complex behaviour we liken them to non-linear oscillators. If this analogy is accepted then the networks in the proceeding discussion can be likened to driven (or forced) non-linear oscillators.

If the input to an ANN is varied over time then—clearly—a description of the network's response requires a time argument. Varying the input of ANNs over time is a current research topic in Sequence Learning [64]. Consider the network shown in Figure 3.10, which we call $A_c$. Furthermore, consider inputting the sequence $S_{in} = ((0, 0), (1, 0), (0, 1), (1, 1), (0, 0), (1, 0), (0, 1), (1, 1))$ into $A_c$ with the rightmost term input at moment $t = 0$, the next rightmost term

[§§]Various asynchronous updating schemes are possible [61]. For example, at each moment we could randomly select a neuron and evaluate its output function.

| | | Output of neurons | | | |
|---|---|---|---|---|---|
| $t$ | $(x_0, x_1)$ | $f_t$ | $g_t$ | $h_t$ | $(y_0, y_1)$ |
| 0 | (0,1) | 0 | 0 | 0 | (0,0) |
| 1 | (0,1) | 0 | 1 | 1 | (1,1) |
| 2 | (0,1) | 0 | 1 | 1 | (1,1) |
| 3 | (0,1) | 0 | 1 | 1 | (1,1) |
| 4 | (0,1) | 0 | 1 | 1 | (1,1) |
| 5 | (0,1) | 0 | 1 | 1 | (1,1) |
| 6 | (0,1) | 0 | 1 | 1 | (1,1) |
| 7 | (0,1) | 0 | 1 | 1 | (1,1) |

**Table 3.1:** The response of $A_b$ to the input $(0, 1)$ over several values of time $t$.

| | | Output of neurons | | | |
|---|---|---|---|---|---|
| $t$ | $(x_0, x_1)$ | $f_t$ | $g_t$ | $h_t$ | $(y_0, y_1)$ |
| 0 | (1,0) | 0 | 0 | 0 | (0,0) |
| 1 | (1,0) | 0 | 0 | 1 | (1,0) |
| 2 | (1,0) | 1 | 0 | 1 | (1,0) |
| 3 | (1,0) | 1 | 0 | 0 | (0,0) |
| 4 | (1,0) | 0 | 0 | 0 | (0,0) |
| 5 | (1,0) | 0 | 0 | 1 | (1,0) |
| 6 | (1,0) | 1 | 0 | 1 | (1,0) |
| 7 | (1,0) | 1 | 0 | 0 | (0,0) |

**Table 3.2:** The response of $A_b$ to the input $(1, 0)$ over several values of time $t$.

input at moment $t = 1$, etc. Table 3.3 shows $A_c$'s response to $S_{in}$. From this table we see that the output of $A_c$ is $S_{out} = (1, 1, 0, 1, 1, 1, 0, 0)$, where the rightmost term is $A_c$'s output at time $t = 0$, the next rightmost term is $A_c$'s output at time $t = 1$, etc. If we treat $A_c$ as a static network and compose its neurons' functions, as we described in Section 3.5.1, then we determine that $A_c$ performs the following mapping: let $A$ denote the input into neuron $\textcircled{f}$, and let $B$ denote the input into neuron $\textcircled{g}$, then—assuming that $A_c$ is a static network—the output from neuron $\textcircled{j}$ is given by $(A \vee B) \vee (B \wedge B) = A \vee B = B \vee A$. That is, the output of $A_c$ should give Inclusive-OR of the input. Termwise evaluation of Inclusive-OR of $S_{in}$ gives $S_{\vee} = (0, 1, 1, 1, 0, 1, 1, 1)$. If we remove the two rightmost terms of $S_{out}$ and remove the two leftmost terms of $S_{\vee}$ then the resulting sequences are identical. This is because $A_c$'s output is termwise Inclusive-OR of the input but information takes two moments of time to 'percolate through' $A_c$.

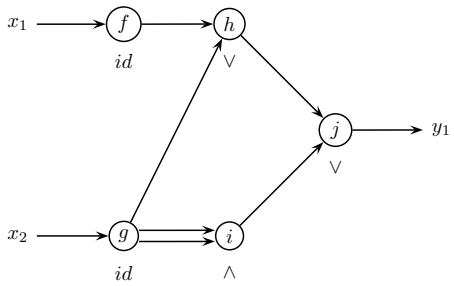



**Figure 3.10:** An ANN that maps $[\mathbb{Z}_2]^2$ to $\mathbb{Z}_2$. The neuron functions $f$ and $g$ both are the identity $id : \mathbb{Z}_2 \to \mathbb{Z}_2$, the neuron functions $h$ and $j$ both are Inclusive-OR, and the neuron function $h$ is logical AND.

| | | Output of neurons | | | | | |
|---|---|---|---|---|---|---|---|
| $t$ | $(x_0, x_1)$ | $f_t$ | $g_t$ | $h_t$ | $i_t$ | $j_t$ | $(y_0, y_1)$ |
| 0 | (1,1) | 1 | 1 | 0 | 0 | 0 | 0 |
| 1 | (1,0) | 1 | 0 | 1 | 1 | 0 | 0 |
| 2 | (0,1) | 0 | 1 | 1 | 0 | 1 | 1 |
| 3 | (0,0) | 0 | 0 | 1 | 1 | 1 | 1 |
| 4 | (1,1) | 1 | 1 | 0 | 0 | 1 | 1 |
| 5 | (1,0) | 1 | 0 | 1 | 1 | 0 | 0 |
| 6 | (0,1) | 0 | 1 | 1 | 0 | 1 | 1 |
| 7 | (0,0) | 0 | 0 | 1 | 1 | 1 | 1 |

**Table 3.3:** The response of $A_c$ to an input that varies over time, $t$.

All non-linear oscillators are forced, but sometimes the driving force may be constant or zero. Similarly, all ANNs have a sequential input, but the input sequence may be constant

over time. If we have an ANN $A$ that is subjected to a constant input sequence then we say that $A$ has a *clamped* input; if $A$ is responding to a varying input sequence then we say that $A$ has a *sequential*, or *non-clamped*, input. Often in the ANN literature the input scheme is not discussed, usually it is implicit that the input is clamped: either because the networks are static, or it is assumed that an appropriate initial time is discovered so that a constant response can be recorded.

Both feedback and sequential input necessitate the consideration of a time argument when describing an ANN. One useful apparatus to model ANNs that uses discrete time is the finite automaton (also called a finite state machine). Minsky describes McCulloch and Pitts networks in terms of finite automata [33, ch. 3]. We use finite automata when we describe Turing's networks in Chapter 4. In the next section we describe finite automata and in Chapter 4 we provide a formal definition.

## 3.6 Cellular Automata

In this section we provide a brief overview of cellular automata; Schiff [62] provides a clear introduction, and Wolfram [65] provides a voluminous treatment of the topic. We introduce cellular automata here for three reasons: it is an area of current AI research, it intersects our research of Turing's ANNs, and Turing did pioneering work in this field (see the next section).

A finite automaton is an idealized type of machine that has a finite number of states and operates with a discrete time scale—from moment to moment. The output of an automaton $A$ at any moment $t$ is entirely dependent on $A$'s input at moment $t-1$ and $A$'s state at moment $t-1$ (see Minisky [33, chap2])[¶¶]. Even though automatons do not deal with continuous time and often they are implemented with a rather small set of possible states, they can produce complex behaviour. Furthermore, often they are an introductory step for the definition of a Turing machine, allowing an investigation into computability.

Cellular automata are a class of finite automata. They are an array of cells with a finite set of states for each cell and a small set of rules that prescribe how the cells' states change from moment to moment. Conway's Game of Life is popular example of cellular automata [62, p94]. In the game of life the array of cells is two dimensional and each cell can have one of two states: alive or dead. Furthermore, all cells are updated (synchronously) at each moment. The rules are chosen to give complex behaviour. Gardner [66, p94] describes simulating the game of life on a *go* board, with the presence of a counter indicating that a cell is alive and the absence of a counter indicating that a cell is dead. Gardner describes the rules as follows:

1. Survivals. Every encounter with two or three neighboring counters survives for the next generation.

[¶¶]Minsky [33, p55] shows that automata and ANNs are equivalent. In Chapter 4 we define a (Turing) ANN using automata, and there we present a precise definition of an automaton.

2. Deaths. Each counter with four or more neighbours dies (is removed) from overpopulation. Every counter with one neighbor or none dies from isolation.
3. Births. Each empty cell adjacent to exactly three neighbors—no more, no fewer—is a birth cell. A counter is placed on it at the next move.

This simple system produces complex behaviour. Cellular automata is an active area of research since a small set of rules can give rise to complex 'emergent' behaviour and this proves to be an accurate way to model many natural phenomena; for example, termite colonies, swarm behaviour, and electrical activity in the brain are modelled with cellular automata [62, chap5].

## 3.7 Turing's Contribution

Alan Turing made great contributions to mathematics, logic, cryptography, and computer science [67][68]. He also contributed to ANNs and cellular automata, and possibly genetic algorithms [69]—all of which are active areas of current AI research. In the technical report *Intelligent Machinery* [1] Turing presented many of his pioneering AI ideas. Copeland [67, p401] writes:

> In [Intelligent Machinery] Turing brilliantly introduced many of the concepts that were later to become central to [AI], in some cases after reinvention by others. These included the logic based approach to problem-solving, and the idea, subsequently made popular by Newell and Simon, that (as Turing put it) 'intellectual activity consists mainly of various kinds of search'. Turing anticipated the concept of a genetic algorithm, in a brief passage concerning what he calls 'genetical or evolutionary search'. 'Intelligent Machinery' also contains the earliest description of (a restricted form of) what Turing was later to call the 'imitation game' and is now known simply as the Turing test. The major part of 'Intelligent Machinery', however, consists of an exquisite discussion of machine learning, in which Turing anticipated the modern approach to AI known as connectionism.

In this section we introduce ANNs that Turing used to investigate his connectionist ideas, and we direct the reader to literature that discusses his other pioneering ideas in AI.

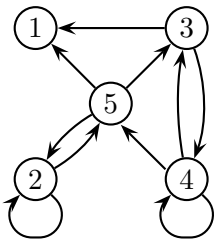
### 3.7.1 A-Type Unorganised Machines

In 1948, seemingly independent of McCulloch and Pitts' ideas [67, p408], Turing introduced the idea of employing networks of interconnected 'units'. In *Intelligent Machinery* Turing wrote the following [67].

> A typical example of an unorganised machine would be as follows. The machine is made up from a rather large number $N$ of similar units. Each unit has two

input terminals, and has an output terminal which can be connected to the input terminals of (0 or more) other units. We may imagine that for each integer $r, 1 \leq r \leq N$, two numbers $i(r)$ and $j(r)$ are chosen at random from $1 \ldots N$. and that we connect the inputs of unit $r$ to the outputs of units $i(r)$ and $j(r)$. All of the units are connected to a central synchronising unit from which synchronising pulses are emitted at more or less equal intervals of time. The times when these pulses arrive will be called 'moments'. Each unit is capable of having two states at each moment. These states may be called 0 and 1. The state is determined by the rule that the states of the units from which the input leads come are to be taken at the previous moment, multiplied together and the result subtracted from 1. An unorganised machine of this character is shown in the diagram below.

| $r$ | $i(r)$ | $j(r)$ |
|---|---|---|
| 1 | 3 | 2 |
| 2 | 3 | 5 |
| 3 | 4 | 5 |
| 4 | 3 | 4 |
| 5 | 2 | 5 |




A sequence of six possible consecutive conditions for the whole machine is:

| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 0 | 1 | 0 |
| 2 | 1 | 1 | 1 | 0 | 1 | 0 |
| 3 | 0 | 1 | 1 | 1 | 1 | 1 |
| 4 | 0 | 1 | 0 | 1 | 0 | 1 |
| 5 | 1 | 0 | 1 | 0 | 1 | 0 |

The behavior of a machine with so few units is naturally very trivial. However, machines of this character can behave in a very complicated manner when the number of units is very large. We may call these A-type unorganised machines. Thus the machine in the diagram is an A-type unorganised machine of 5 units. The motion of an A-type machine with $N$ units is of course eventually periodic, as in any determined machine with finite capacity. The period cannot exceed $2^N$ moments, nor can the length of time before the periodic motion begins. In the example above the period is 2 moments and there are 3 moments before the periodic motion begins. $2^N$ is 32.

In today's language, Turing's A-type unorgansied machines are Boolean dynamic ANNs that are synchronously updated. Copeland and Proudfoot [70, p364] point out that "any Boolean operation can be formed by a circuit consisting entirely of NAND units. Thus any such operation can be performed by an A-type machine". From the above excerpt from

*Intelligent Machinery* it is not clear how information is entered into these machines. It may be that information is entered via the machine's initial state. We examine this further next when we introduce Turing's B-type unorganised machines and in Chapter 5 when we introduce our implementation of A-type unorganised machines.

### 3.7.2 B-Type Unorganised Machines

Turing suggested a mechanism for training A-type unorganised machines, namely these machines were set to a particular configuration so that the interconnections between some nodes could be switched on or off by setting the state of other nodes. Turing called these configurations B-type unorganised machines. Consider the A-type unorganised machine shown in Figure 3.11(a). The output from node $B$ (say) has three types of response: oscillatory behaviour if $(A, B) = (0, 0)$ or if $(A, B) = (1, 1)$, a constant output of 0 if $(A, B) = (1, 0)$, or a constant output of 1 if $(A, B) = (0, 1)$. This behaviour is employed to construct a switching machine that depends on its initial state, shown in Figure 3.11(c). Copeland calls these machines *connection modifiers* [67, p406], and Turing illustrated them with the symbol shown in Figure 3.11(d). Turing defined B-type unorganised machines as A-type unorganised machines such that there is a connection modifier between each node. For example Figure 3.12 illustrates a B-type unorganised machine. So B-types are machines that can be constructed in an unorganised fashion and can be trained by modifying the states of the modifiers' nodes***. If Turing's B-type unorganised machine was implemented in hardware then its architecture offers a means of reconfiguring it without physically changing the device.

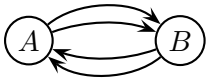
### 3.7.3 Further Invention

In the early 1950's Turing conducted pioneering research on modelling biological systems using simple rules [62, p124]. Copeland and Proudfoot [69, p103] write:

> During [the early 1950's], Turing achieved the distinction of being the first to engage in the computer assisted exploration of nonlinear dynamical systems. His theory used nonlinear differential equations to express the chemistry of growth.

Turing presented this seminal work in [71]. Further detail is found in [67, ch.15] and [68, pp477-496]. Mindful of this work we suggest that evolutionary algorithms are an appropriate interpretation of Turing's 'genetical search' for A-type machines.

***Copeland and Proudfoot [70] show that a B-type is not universal but this can be remedied by having two connection modifiers between each node. Teuscher [2, p29] introduces a similar B-type architecture that accepts input to switch its modifiers rather than relying on the modifiers' initial state to determine the interconnections within the machine.

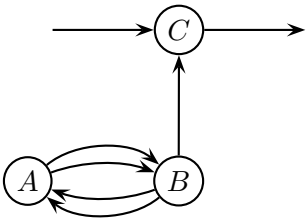



| moment | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| $A$'s state | 0 | 1 | 0 | 1 | 0 | 1 |
| $B$'s state | 0 | 1 | 0 | 1 | 0 | 1 |

| moment | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| $A$'s state | 0 | 0 | 0 | 0 | 0 | 0 |
| $B$'s state | 1 | 1 | 1 | 1 | 1 | 1 |

(a) Consider the A-type unorganised machine above. From the above tables we see how the state of node $B$ depends on the initial states of nodes $A$ and $B$.


(b) Consider the response of node $C$ if one of its values is assigned constant value.




(c) A connection modifier.

(d) A connection modifier symbol.

**Figure 3.11:** Illustrating the components of a B-type's connection modifier.

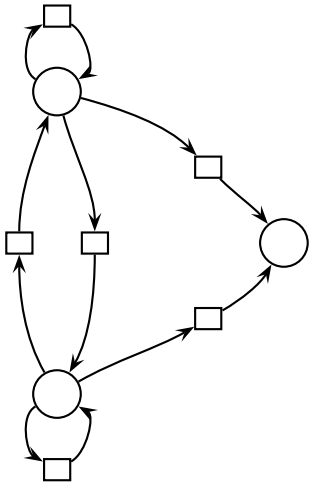


**Figure 3.12:** An example of a B-type unorganised machine.

### 3.7.4 Exploring Turing's Connectionist Ideas

Artificial neural networks have found wide application and are an active area of research (see Section 3.5), yet few researchers have continued Turing's pioneering work on ANNs. The most notable continuation of his work is research conducted by Teuscher [2]. Teuscher experimented with A-types with fixed input states; for instance, he used A-types in this manner to solve basic pattern classification tasks. Teuscher employed EAs to train Turing's networks. Teuscher used linear chromosome representations of B-types when he implemented EAs on Turing networks. He used B-types with lists that prescribed whether each connection modifier was in a 'connected' or 'disconnected' state [2, p88]. These lists give linear chromosomes for Teuscher's A-types.

Today Turing's A-types can be considered a special class of Random Boolean Networks [2, p24]. Random Boolean Networks are simple discrete dynamical systems that are capable of complex behaviour; consequently, they are useful for modelling complex systems such as gene regulation mechanisms in biology and the internet [72]. Teuscher investigated the non-linear dynamics of A-types [2, ch 5] [3]. Recently, Bull [73], and Bull and Preene [74] have investigated the evolution of Turing's A-type machines, and they consider this in the context of discrete dynamical systems.

## 3.8 Conclusion

Machine learning is vast field. Currently there are many useful technological applications, and research promises progress towards artificial intelligence. In this chapter we presented a small sample of machine learning. In particular we presented a brief overview of evolutionary computing and artificial neural networks. Hopefully this chapter whet the reader's appetite. Turing's ideas on machine learning are historically important. In the latter chapters of this thesis we use his ideas on artificial neural networks and evolutionary computing in our research.

# 4 Interpreting A-types as Finite State Machines

## 4.1 Aims of this Chapter

In this chapter we interpret Turing's A-Type machines as finite state machines (FSMs). Our research (which we detail in latter chapters) uses A-type machines as dynamically driven ANNs. That is, our A-types accept sequential input and return sequential output. Turing's definition does not explicitly specify how A-types can be implemented as machines that accept input strings and return output strings. We invest considerable effort making our interpretation of Turing's A-types precise. We adopt language from graph theory and employ the definition of a finite state machine to make our presentation precise. This is useful because we require careful consideration of the discrete and dynamic nature of our A-types. We introduce a second type of basic neuron to our A-type networks. This neuron enables the the synchronization of information that flows through the network.

In this chapter we formally present a FSM, define two basic types of FSM, and define A-type machines as machines composed of these basic machines. In the next chapter we explain how we use these machines to process information—we drive them with sequential input.

## 4.2 Finite State Machines

Here we present a formal definition of a FSM. Our approach has been particularly influenced by Minsky's description of McCulloch-Pitts networks [33, ch.3]. We also use Arbib [75] which provides a very mathematical treatment of the subject. A FSM is a computing machine that has input stimuli, internal states, and output responses. The input stimulus, internal state and output response are instantaneously updated at regular intervals—we can imagine that this happens once every second. The internal state at a given instant is a function of both the input stimuli and the internal state at the previous moment. Similarly, the output response at a given instant is a function of both the input stimulus and the internal state at the previous moment. Figure 4.1 gives a pictorial representation of a FSM, this may help the reader conceptualize the following definition.

**Definition 4.1** (Finite State Machine)**.** A *finite state machine* is a quintuple $M = (X, Y, Q, f, g)$ where

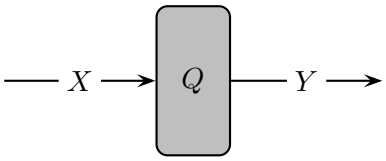



**Figure 4.1:** A pictorial representation of a finite state machine. Note that $X$ is the set of possible input sets, $Q$ is the set of possible states, $Y$ is the set of possible output sets.

1. $X$ is a finite set, the set of *inputs*;
2. $Y$ is a finite set, the set of *outputs*;
3. $Q$ is a finite set, the set of *states*;
4. $f : Q \times X \to Q$, the *next state function*;
5. $g : Q \times X \to Y$, the *next output function*.

□

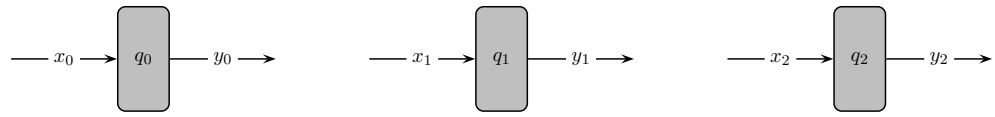
The above definition is the same as that given by Arbib [75, p57]. After presenting this definition Arbib proceeds to provide the following explanation. Note that when we reproduce Arbib's quote we change his notation to match our own.

> We interpret this formal quintuple as being a mathematical description of a machine which, if at a time $t$ is in a state $q$ and receives input $x$, will at time $t + 1$ be in a state $f(q, x)$ and will emit an output $g(q, x)$.

We formalize the notion of discrete time steps by determining a time function that maps the non-negative integers to elements of a FSM. We then define how a next state function and a next output function act on the elements of a FSM with respect to the time function. We present this in the following definition.

**Definition 4.2** (Moments)**.** Given a finite state machine $M = (X, Y, Q, f, g)$ and a triple $\alpha = (x_\alpha, y_0, q_0)$ consisting of the following: a sequence of inputs $x_\alpha = (x_0, x_1, x_2, \ldots)$ where each term is a member of $X$; an initial state $q_0 \in Q$; and an initial output $y_0 \in Y$. We define a map $T_\alpha : \mathbb{N}_0 \to X \times Y \times Q$ by $T_\alpha(t) = (x_t, y_t, q_t)$ where $t \in \mathbb{N}_0$, and for $t \geq 1$ $q_t = f(q_{t-1}, x_{t-1})$ and $y_t = g(q_{t-1}, x_{t-1})$. We call $T_\alpha$ the *time function* of $M$ with respect to $\alpha$ and we call each $t \in \mathbb{N}_0$ a *moment*.

□




(a) At the moment $t = 0$ assign $x_0$, assign $q_0$, and assign $y_0$.

(b) At the moment $t = 1$ assign $x_0$, determine $q_1 = f(q_0, x_0)$, and determine $y_1 = g(q_0, x_0)$.

(c) At the moment $t = 2$ assign $x_2$, determine $q_2 = f(q_1, x_1)$, and determine $y_2 = g(q_1, x_1)$.

**Figure 4.2:** A picture of a FSM in each of its first three moments.

Figure 4.2 illustrates of the way a FSM changes through discrete moments.

Definition 4.1 is very general* and, even with the notation presented in Definition 4.2, FSMs are rather abstract. Recall that A-type machines are discrete dynamic ANNs, furthermore in the next chapter we define them with sequential input. So discrete time is integral in these networks and FSMs are an appropriate and conventional model to describe such machines. To clarify how we use FSMs to describe our A-type machines we introduce the following, more specific, idea of a Boolean finite state machine.

**Definition 4.3** (Boolean Finite State Machine)**.** A *Boolean finite state machine* (BFSM) is a quintuple $M = (X, Y, Q, f, g)$ such that $M$ is a finite state machine; the set of *inputs* $X$ is $(\mathbb{Z}_2)^m$ for some integer $m$; the set of *outputs* $Y$ is $(\mathbb{Z}_2)^n$ for some integer $n$; and the set of *states* $Q = (\mathbb{Z}_2)^p$ for some integer p. We call $m$ the *input dimension* of $M$, and $n$ the *output dimension* of $M$.

□

We provide two examples of simple BFSMs. These examples are chosen to serve a dual purpose. First, they aid our presentation of BFSMs. Second, they form part of our definition of an A-type machine.

## 4.3 Delay Machines

Our first example is the delay machine. It has a single input, a single state, and a single output for each moment. The initial state is 0 and at any later moment it is simply the input at the previous moment. Furthermore, the output at a given moment is the value of the

*Often this general description of a FSM is presented and then extended to introduce the Turing machine [33]

state at that same moment. If we imagine information flowing, one bit per moment, through a delay machine then information is not altered except that it is delayed by one moment. We give pictorial representation of a delay machine in Figure 4.3, and we provide a formal definition next.

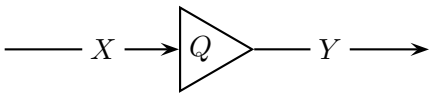
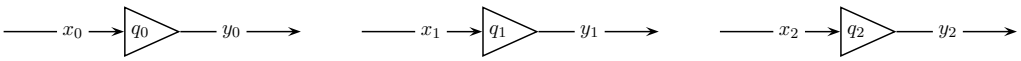



**Figure 4.3:** A picture of a delay machine. The set of possible inputs, $X$, is $(\mathbb{Z}_2)^1$. The set of possible states, $Q$ is $(\mathbb{Z}_2)^1$. The set of possible outputs, $Y$, is also $(\mathbb{Z}_2)^1$.




(a) At the moment $t = 0$ assign $x_0 \in (\mathbb{Z}_2)^1$, assign $q_0 = 0$, and assign $y_0 = 0$.

(b) At the moment $t = 1$ assign $x_1 \in (\mathbb{Z}_2)^1$, determine $q_1 = x_0$, and determine $y_1 = x_0$.

(c) At the moment $t = 2$ assign $x_2 \in (\mathbb{Z}_2)^1$, determine $q_2 = x_1$, and determine $y_2 = x_1$.

**Figure 4.4:** A picture of a delay machine in each of its first three moments.

**Definition 4.4** (Delay Machine)**.** A *delay machine* is a Boolean finite state machine $(X, Y, Q, f, g)$ such that

1. $X$ is $\mathbb{Z}_2$;
2. $Y$ is $\mathbb{Z}_2$;
3. $Q$ is $\mathbb{Z}_2$;
4. $q_0 = 0$, and $f(q_t, x_t) = x_{t-1}$ for all $t > 0$;
5. $g = f$.

□

Definition 4.4 prescribes a BFSM that accepts one bit per moment and outputs that bit in the next moment. We constructed this definition so that a delay machine simply 'staggers information flow' by one moment. For definiteness our definition requires detail of the next state function $f$. We could put an arbitrary function $f$ in Definition 4.4 and the next output

function $g$ would still serve our purpose of staggering information flow. We choose to make $f$ equal $g$ because when we think about the delay machine it is helpful to imagine the output as simply its current state[†].

## 4.4 Nand Machines

Our second example is the nand machine. It has an input dimension of two and an output dimension of unity. The value of the output at any given moment is the logical NAND of the two inputs at the previous moment. We give a pictorial representation of a nand machine in Figure 4.5. Next we provide a formal definition of a nand machine.

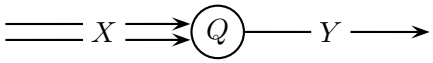
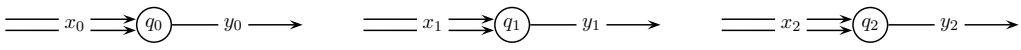



**Figure 4.5:** A picture of a nand machine. The set of possible inputs, $X$, is $(\mathbb{Z}_2)^2$. The set of possible states $Q$ is $(\mathbb{Z}_2)^1$. The set of possible output sets $Y$ is $(\mathbb{Z}_2)^1$.




(a) At the moment $t = 0$ assign $x_0 \in (\mathbb{Z}_2)^2$, assign $q_0 = 0$, and assign $y_0 = 0$.

(b) At the moment $t = 1$ assign $x_1 \in (\mathbb{Z}_2)^2$, determine $q_1 = NAND(x_0)$, and determine $y_1 = NAND(x_0)$.

(c) At the moment $t = 2$ assign $x_2 \in (\mathbb{Z}_2)^2$, determine $q_2 = NAND(x_1)$, and determine $y_2 = NAND(x_1)$.

**Figure 4.6:** A pictorial representation of a nand machine in each of its first three moments.

**Definition 4.5** (Nand Machine)**.** A *nand machine* is a Boolean finite state machine $(X, Y, Q, f, g)$ such that

1. $X$ is $(\mathbb{Z}_2)^2$;
2. $Y$ is $\mathbb{Z}_2$;
3. $Q$ is $\mathbb{Z}_2$;

[†]We have chosen this definition to be in keeping with how we describe our A-type machines. The states of nodes in an A-type change from moment to moment and we will read off the state of a set of nodes to generate the output for that moment. This is detailed later but we note it here because it motivates the form of the above definition.

4. $f(q_t, x_t) = \begin{cases} 1 & \text{if } x_t = (1,1) \\ 0 & \text{otherwise} \end{cases}$

   where $q_0 = 0, q_t \in (\mathbb{Z}_2)^1$ for all $t \geqslant 1$, and $x_t \in (\mathbb{Z}_2)^2$ for all $\geqslant 0$.

5. $g = f$.

□

Note that for both delay machines and nand machines the output value at any given moment is a function only of the input of the previous moment—it is independent of the machines' state at the previous moment. In this sense, both delay machines and nand machines are very simple BFSMs. In terms of retrieving the desired output, our choice of the next state function, $f$, in Definition 4.5 is arbitrary. This was also the case in Definition 4.4 and in both cases we define $f$ to be equal to the next output function, $g$, to help us visualise the operation of our machines.

## 4.5 A-type Machines

In this section we define an A-type machine as a BFSM composed of BFSMs. The definition of a directed graph (given in Section 2.2) helps us concisely prescribe how we employ Turing's A-type machines. Specifically, we define an A-type machine to be composed solely of delay machines and nand machines, and all machines are updated synchronously. The interconnection of the constituent machines is presented as a directed graph.

**Definition 4.6** (A-type Machine)**.** An *A-type machine* is a septuple $M = (N, E, I, O, init, term, \delta)$ such that the following conditions hold.

The set $N$ is a finite set, every element of which is either a delay machine or a nand machine. We call $N$ the set of *nodes*.

The set $I$ is a subset of $N$. We call $I$ the set of *input nodes*.

The set $O$ is a subset of $N$. We call $O$ the set of *output nodes*.

We call the set $N - I - O$ the set of *internal nodes*.

The source of each node's input is prescribed by the directed graph $(N, E, init, term)$. Every input node has an indegree of zero. Every non-input node that is a delay machine has an indegree of one. Every non-input node that is a nand machine has an indegree of two. No arrow can both exit an input node and enter an output node. Furthermore, the set of internal nodes must contain at least one element.

The A-type $M$ gives rise to a Boolean finite state machine $(X, Y, Q, f, g)$ where

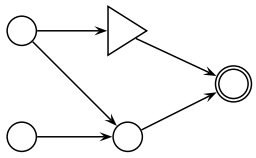


**Figure 4.7:** A picture of our A-type $M$. The nand machines are represented as circles and the delay machines are represented as triangles. The input nodes are represented by circles that have no incoming arrows, and the output node is shown as the node that is two concentric circles.

1. $X$ is $(\mathbb{Z}_2)^{|I|}$;
2. $Y$ is $(\mathbb{Z}_2)^{|O|}$;
3. $Q$ is $(\mathbb{Z}_2)^{|N|}$;
4. The next state function, $f$, is such that for every moment $t \geqslant 0$ the state of the $i$th input node is assigned to the $i$th entry of $x_t \in X$. At moment $t = 0$ every non-input node is assigned the state 0. For each moment $t > 0$ the state of every non-input node is updated according to the node type. That is, for each non-input node $\mathcal{N}_i$ that is a delay machine there is an edge from some node $\mathcal{N}_j$ to $\mathcal{N}_i$. For each moment $t > 0$ the input of $\mathcal{N}_i$ is the output of $\mathcal{N}_j$ at the moment $t-1$; hence the state of $\mathcal{N}_i$ at moment $t$ is the state of $\mathcal{N}_j$ at the moment $t-1$. Similarly, for each non-input node $\mathcal{N}_u$ that is a nand machine there are two edges sourced from (not necessarily distinct) nodes $\mathcal{N}_v$ and $\mathcal{N}_w$. For each moment $t > 0$ the input of $\mathcal{N}_u$ is constructed by taking the output of $\mathcal{N}_v$ at moment $t-1$ and the output of $\mathcal{N}_v$ at moment $t-1$ as a pair; hence the state of $\mathcal{N}_u$ at moment $t$ is the NAND of this pair.
5. The next output function, $g$, is the next state function, $f$, restricted to the output nodes. That is, the output at moment $t$, $y_t$, is the $|O|$-tuple in which the $i$th entry is the state of the $i$th output node at moment $t$.

The component $\delta$ is a non-negative integer, which we call the *delay*.

□

For example, consider the following A-type machine, $M$, that has five nodes in total, an input dimension of two and an output dimension of one. One of $M$'s internal nodes is a delay machine, the other is a nand machine. The internal node that is a delay machine is the target of an arrow that is sourced from one of the input nodes. The other internal node is the target for two arrows, each of which is sourced from a different input node. There is an arrow from the internal node that is a delay machine to the output node. Also there is

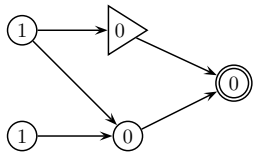
an arrow from the internal node that is a nand machine to the output node. The delay of $M$ is two. In spite of $M$ being a relatively simple A-type the above description is long and difficult to follow. This is because we are giving a written description of $M$'s graph. As with graphs, it is useful to use pictures to represent A-types. Figure 4.7 shows a picture of $M$, the nand machines are represented as circles, the delay machines as triangles and the output nodes as double lined circles[‡]. The input nodes are the two nodes that have an indegree of zero. From the diagram we can see that $M$ has two input nodes so the input set must be $X = (\mathbb{Z}_2)^2$. Since $M$ has a single output node the output set must be $Y = (\mathbb{Z}_2)^1$. Let the first three input pairs be $x_{t=0} = (1, 1)$, $x_{t=1} = (0, 1)$, $x_{t=2} = (1, 0)$. Consequently, the first three output values are $y_{t=0} = 0$, $y_{t=0} = 1$ , $y_{t=0} = 1$. This may be more apparent if the state of the constituent machines of $M$ are displayed for each moment; we show this in Figure 4.8.

Note that the states of the input nodes are assigned the input values (in the correct order) for each moment. The output value for each moment is determined by collecting the value of the output node at that moment. Also note that the initial state of $M$ is such that the state of every non-input node is zero.

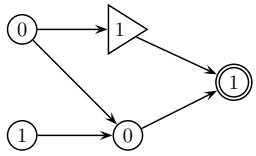
If we compare our definition with Turing's definition of an A-type machine (see Section 3.7) then there are differences. Turing did not mention nodes like our delay machines. Turing's definition does not explicitly specify how A-types can be implemented as machines that accept input strings and return output strings. We introduce delay machines to allow the synchronization of information flow through an A-type. If we consider the interconnecting wires of Turing's A-type machines to have a finite propagation speed then we can introduce delays by adjusting the relative lengths of a machine's wires. Another difference is that our A-type machines have input nodes. This is necessary because we input data via a sequential input; whereas, Turing was not explicit about how data was input—we assume that data is entered via the initial states of all nodes. If Turing's machines are used in this fashion then they must be clamped networks: without sequential input. Another difference is that our A-type machines have a delay $\delta$. If we re-examine the previous example we find no use of $\delta$. We only require $\delta$ when we implement A-types to process information. Teuscher [2, p31] presents a definition of an A-type machine that has input and output nodes. For his networks to process sequential input they require two clock speeds. The ratio of these two speeds is $\delta$. We explain this further in Section 5.3.6. We mention it here to emphasize that the inclusion of $\delta$ in our definition of an A-type is necessary when we employ A-types to process information.

Finally, note that in the above definition we specify the input to a nand machine as an ordered pair of the output of two nodes. The order of the input to a nand machine is inconsequential since NAND is symmetric with respect to its input variables. So, although we specify a pair, in this case, the order is not important.

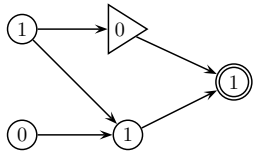
[‡]Our definition of an A-type allows an output node to be a delay machine. In practice, we rarely consider this configuration but if we do we will display such an output node as a double lined triangle.




(a) The A-type $M$ at $t = 0$. The input pair is $x_0 = (1, 1)$ and this prescribes the state of the two input nodes. The state of each of the non-input nodes is 0. Since the output node has the state 0 the output of $M$ is 0.




(b) The A-type $M$ at $t = 1$. The input pair is $x_0 = (0, 1)$ and this prescribes the state of the two input nodes. The state of each non-input node $\mathcal{N}$ is a function (specifically, the next state function) of the state at $t = 0$ of the source nodes of $\mathcal{N}$. For example, the state of the output node is $0 \,\overline{\wedge}\, 0 = 1$, which is the NAND of the states of the two internal nodes at $t = 0$. Similarly, the state of the internal nand machine is the NAND of the previous two input values; namely $1 \,\overline{\wedge}\, 1 = 0$. Finally, note that since the output node has the state 1 the output of $M$ is 1.




(c) The A-type $M$ at $t = 2$. The input pair is $(1, 0)$ and this prescribes the state of the two input nodes. The state of each non-input node $\mathcal{N}$ is a function (the next state function) of the state at $t = 1$ of the source nodes that enter $\mathcal{N}$. For example, the state of the output node is $1 \,\overline{\wedge}\, 0 = 1$, which is the NAND of the states of the two internal nodes at $t = 1$. Finally, note that since the output node has the state 1 the output of $M$ at moment $t = 2$ is 1.

**Figure 4.8:** A sequence of three pictures demonstrating how the the A-type $M$ changes over three moments.

# 5 Employing A-types to Process Information

## 5.1 Aims of this Chapter

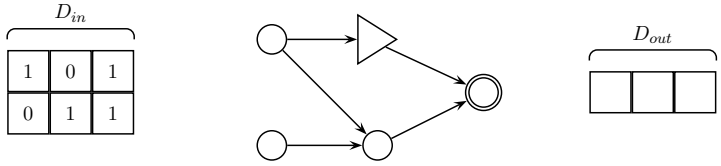
In this chapter we explain how we use our A-types to accept and output sequential data. Our definitions from the previous chapter enable us to precisely describe the discrete operation of our A-types as they process information.

## 5.2 Generating Data Packets

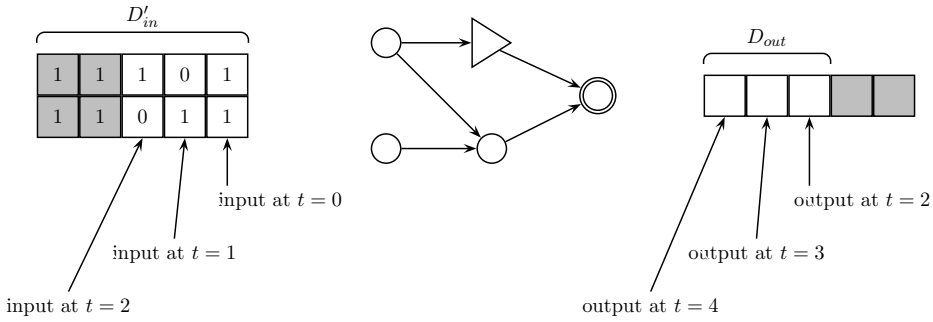
In the previous chapter we defined A-types to have input nodes and output nodes whose states, in general, vary over time. For each moment the states of the input nodes must be assigned. Initially the state of each output node is 0 and for later moments the states of output nodes are a function of the particular A-type and its history. So we can enter information into an A-type by assigning the states of the input nodes moment by moment. Also, we can collect information from an A-type by recording the states of the output nodes moment by moment. Thus we can use an A-type as a dynamic ANN with sequential input (see Section 3.5). Recall that such an ANN is analogous to a dynamically driven oscillator. With an oscillator there may be a time lag between entering a driving signal and receiving a response. Similarly, it may take several moments for information to 'percolate through' an ANN before the output is deemed desirable. Our A-types were defined with a delay, $\delta$, to specify the number of moments that must elapse between first entering input and first collecting output.

We employ data packets (see Section 2.3) to describe how we enter sequences of information into, and collect sequences of information from, A-types. Informally, given an A-type $A$ with a delay $\delta$ we can input a data packet $D_{in}$ and generate a data packet $D_{out}$. We do this by entering columns (moment by moment, right to left) of $D_{in}$ into $A$, and after $\delta$ moments we begin constructing $D_{out}$ by assigning (moment by moment, right to left) the columns of $D_{out}$ to be $A$'s output states. We formally present this in the following definition.

**Definition 5.1** (Generating data packets)**.** Let $m, n, p, q$ denote some positive integers. Consider an A-type machine $A$ that has an input dimension $m$, an output dimension $p$, and a delay $\delta$. Also, let $D_{in}$ denote some $m \times n$ data packet. Consider the $p \times q$ data packet $D_{out}$ that is returned by the following algorithm.




(a) The data packet for the input $(x_0 = (1,1), x_1 = (0,1), x_2 = (1,0))$, the A-type $A$ (with a delay $\delta = 2$), and a skeleton of the output data packet.




(b) The data packet with two shaded columns of shunt bits, the A-type $A$, and a skeleton of the output data packet. We append two shaded cells to the output data packet to display the discarded states of the output node of $A$ for the first two moments. The shaded cells are not part of the output data packet.

**Figure 5.1:** Constructing an input data packet from the set of inputs used in Figure 4.8.

The input data packet $D_{in}$ is modified and we denote the resulting data packet $D'_{in}$. If $\delta + q > n$ then we elongate $D_{in}$ as follows. We copy the leftmost column of $D_{in}$ to produce a $m \times 1$ vector. This vector is appended to the left of $D_{in}$, and this process is repeated $\delta$ times. Otherwise, $\delta + q \leq n$, then $D'_{in}$ is only the rightmost $(\delta + q)$ columns of $D_{in}$.

At moment t = 0 the states of $A$'s input nodes are assigned the values of the rightmost column of $D'_{in}$ (in the correct order). At moment t = 1 the states of $A$'s input nodes are assigned the values of the next rightmost column of $D'_{in}$. This process is repeated so that $D'_{in}$ prescribes $A$'s input for the first $(\delta + q)$ moments.

At moment t = $\delta$ the rightmost column of $D_{out}$ is assigned the states of the output nodes (in the correct order) of $A$ for that moment. At moment t = $\delta + 1$ the next rightmost column of $D_{out}$ is assigned the states of the output nodes of $A$ for that moment. This process is repeated until all $q$ columns of $D_{out}$ have been assigned values.

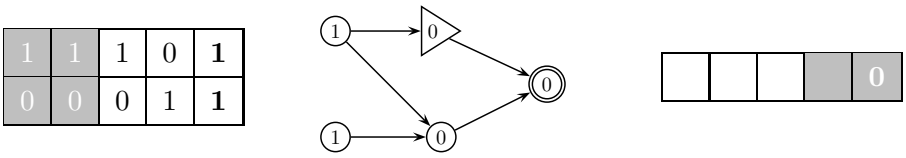
We say that $A$ with input $D_{in}$ *generates* $D_{out}$.

□

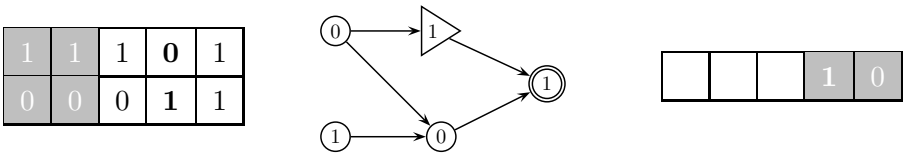
Let us recycle the example in Section 4.5. In figure 5.1(a) we show an input data packet that has the three input pairs: the rightmost column gives the input $x_{t=0} = (1, 1)$, the middle column gives the input $x_{t=1} = (0, 1)$, and the leftmost column gives the input $x_{t=2} = (1, 0)$. We consider an output data packet with three columns. Because $A$'s output dimension is unity the output data packet must be a $3 \times 1$ matrix—we illustrate this as a $3 \times 1$ array of empty boxes. When we generate our output packet the input data packet will have two columns joined to its left because the delay of $A$ is $\delta = 2$. This elongated input data packet, the A-type $A$, and the skeleton of the expected output data packet is shown in Figure 5.1(b). Notice that the extra two columns in the elongated input data packet have the values of the leftmost column of the original input data packet. These extra columns are shaded to emphasize the fact that these bits only serve to 'shunt' the other input bits through $A$. Because the output data packet has three columns and $A$ has a delay of two moments we require five moments to generate the output data packet. A delay of two moments means that the first two states of the output node are discarded. These two bits are placed in the shaded boxes to the right of the array of three boxes that represent the output data packet. In Figure 5.2 we illustrate how $A$ changes as the output packet is generated over these five moments.

## 5.3 Representing Functions

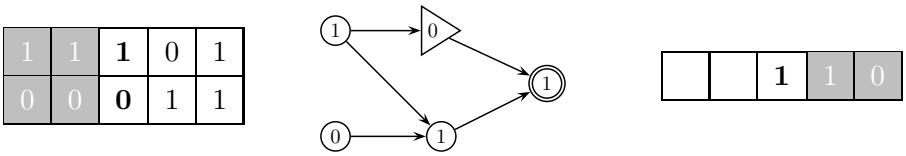
Recall that in Section 3.3 we restricted our definition of learning to a search for a solution function that approximates a given concept function. In latter chapters we apply learning



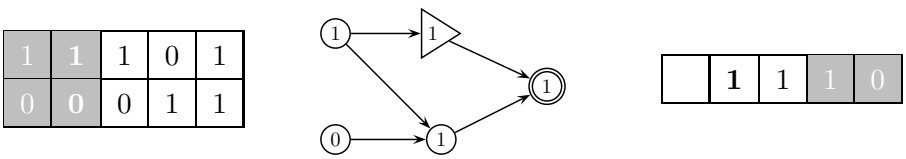

(a) The A-type at $t = 0$. The input nodes of $A$ are assigned the values of the rightmost column of the input data packet (shown in bold). The non-input nodes are initialised to zero. The output value doesn't contribute to the output data packet so we place it in a shaded cell.



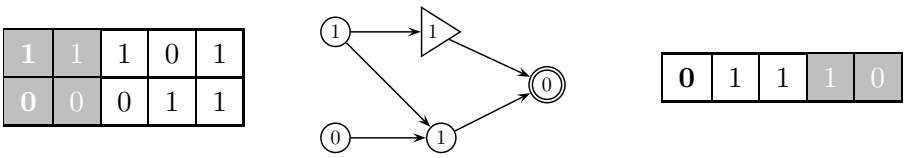

(b) At $t = 1$. The input nodes of $A$ are assigned the values of the second rightmost column of the input data packet. Again, the output value doesn't contribute to the output data packet so we place it in a shaded cell.




(c) At $t = 2$. The input nodes of $A$ are assigned the values of the third rightmost column of the input data packet. The delay of $M$ is two so the output value is the rightmost (least significant) entry of the output data packet (shown in bold).




(d) At $t = 3$. The input nodes of $A$ are set to the first column of 'shunt bits'. Again, the state of the output node contributes to the output data packet.




(e) At $t = 4$. Again the input nodes of $A$ are set to a column of 'shunt bits'. Again, the state of the output node contributes to the output data packet.

**Figure 5.2:** How the A-type $A$ generates an output data packet from the input data packet given in Figure 5.1(a).

algorithms to A-types. To do this we introduce the notion of an A-type *representing* a function. Learning then becomes the task of searching for A-types that represent a given concept function. The functions that we consider are maps between sets of data packets. We describe an A-type representing a function in terms of the set of output data packets that an A-type generates given a set of input packets. We formalize this in the following definition.

**Definition 5.2** (A-types representing functions)**.** Let $M[k, l]$ denote the set of all $k \times l$ data packets. Consider two sets $X \subseteq M[m, n]$, $Y \subseteq M[p, q]$ for some $m, n, p, q \in \mathbb{Z}^+$, and consider the function $f : X \to Y$. We say that an A-type $A$ *represents* $f$ if for every data packet $x \in X$ when $x$ is an input then $A$ generates $f(x)$.

□

Next we present examples of A-types that represent simple Boolean functions. Note that we introduced Boolean functions in Section 2.3). There we defined Boolean functions, clamped Boolean functions, and columnwise Boolean Functions. We devised these definitions to have the following two properties. First, an A-type that represents a clamped Boolean function operates as an ANN with clamped input. Second, an A-type that represents a columnwise Boolean function operates as an ANN with sequential input.

We now present five concrete examples of A-types that represent simple Boolean functions. In addition to their pedagogical value, these examples provide tests for our implementation of A-types into a computer program. We do this by specifying an A-type and appropriate input data packet, and then checking that the expected output data packet are returned—which, in these simple cases, we verify by hand.

For each of the following five examples we display a truth table that defines a Boolean function $f$ and we display an A-type that represents $f$. Again, we show shunt bits in shaded cells appended to the input data packets. Also, we show the discarded bits (the states of the output nodes before $\delta$ moments have elapsed) in shaded cells appended to the output data packet.

#### A Unary operation: Identity

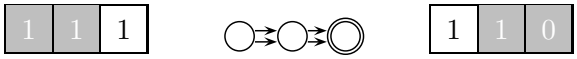
Our first example is unary Boolean identity $id : \mathbb{Z}_2 \to \mathbb{Z}_2$. The first two columns in Table 5.1 define this function and the final two columns demonstrate how identity can be achieved solely with NAND operations. From this we construct an A-type that represents identity, which we illustrate in Figure 5.3.

#### A Unary operation: Negation

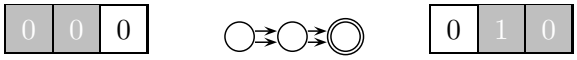
Our second example is unary Boolean negation $\neg : \mathbb{Z}_2 \to \mathbb{Z}_2$. The first two columns in Table 5.2 define this function and the final three columns demonstrate how negation can be achieved solely with NAND operations. From this we construct an A-type that represents

| $A$ | $id(A)$ | $A \barwedge A$ | $(A \barwedge A) \barwedge (A \barwedge A)$ |
|---|---|---|---|
| 1 | 1 | 0 | 1 |
| 0 | 0 | 1 | 0 |

**Table 5.1:** A truth table that suggests how we can represent identity. The first two columns define identity. The final two columns show how we can represent identity in an expression involving only NAND operations.




(a) $id(1) = 1$




(b) $id(0) = 0$

**Figure 5.3:** An A-type $A$, with delay $\delta = 2$, that represents the identity function $id : \mathbb{Z}_2 \to \mathbb{Z}_2$. Each subfigure illustrates a pair $(x, id(x))$ from this function: using an input data packet that corresponds to $x$, $A$ generates a data packet that corresponds to $id(x)$.

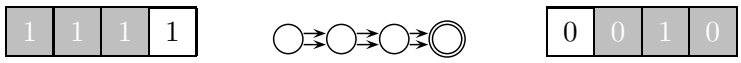
negation, which we illustrate in Figure 5.4. Note that although the third column of Table 5.2 describes negation it does not suggest a means of constructing a valid A-type, because our A-types require at least one internal node.

| $A$ | $\overline{A}$ | $A \barwedge A$ | $(A \barwedge A) \barwedge (A \barwedge A)$ | $((A \barwedge A) \barwedge (A \barwedge A)) \barwedge ((A \barwedge A) \barwedge (A \barwedge A))$ |
|---|---|---|---|---|
| 1 | 0 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |

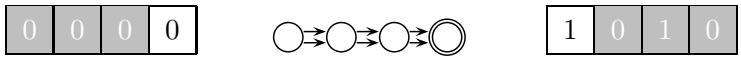
**Table 5.2:** A truth table that suggests how we can represent negation. The first two columns define negation. The final three columns show how we can represent negation in an expression involving only NAND operations.

### A Binary Operation: Inclusive-OR

Our third example is Inclusive-OR $\vee : [\mathbb{Z}_2]^2 \to \mathbb{Z}_2$. The first three columns in Table 5.3 define Inclusive-OR. The final three columns of Table 5.3 demonstrate how Inclusive-OR can be achieved solely with NAND operations. From this we construct an A-type that represents Inclusive-OR, which we illustrate in Figure 5.5.




(a) $\neg(1) = 1$

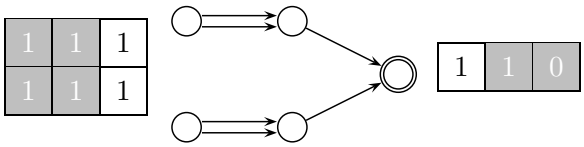



(b) $\neg(0) = 0$

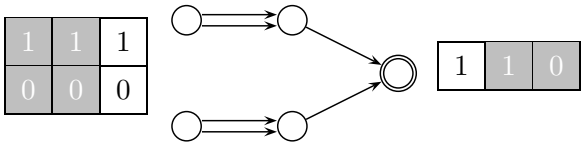
**Figure 5.4:** An A-type $A$, with delay $\delta = 3$, that represents the negation function $\neg : \mathbb{Z}_2 \to \mathbb{Z}_2$. Each subfigure illustrates a pair $(x, \neg(x))$ from this function: using an input data packet that corresponds to $x$, $A$ generates a data packet that corresponds to $\neg(x)$.

| $A$ | $B$ | $A \vee B$ | $A \mathbin{\overline{\wedge}} A$ | $B \mathbin{\overline{\wedge}} B$ | $(A \mathbin{\overline{\wedge}} A) \mathbin{\overline{\wedge}} (B \mathbin{\overline{\wedge}} B)$ |
|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 1 | 1 |
| 1 | 0 | 1 | 1 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 | 1 |
| 0 | 0 | 0 | 1 | 0 | 0 |

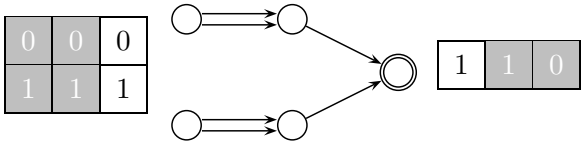
**Table 5.3:** A truth table that suggests how we can represent Inclusive-OR. The first three columns define Inclusive-OR. The final three columns show how we can represent Inclusive-OR in an expression involving only NAND operations.



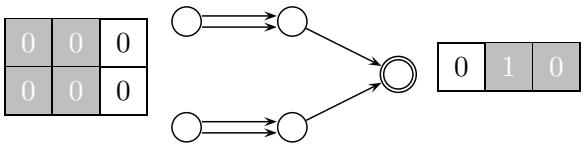

(a) $1 \vee 1 = 1$




(b) $1 \vee 0 = 1$




(c) $0 \vee 1 = 1$




(d) $0 \vee 0 = 0$

**Figure 5.5:** An A-type $A$, with delay $\delta = 2$, that represents Inclusive-OR $\vee : [\mathbb{Z}_2]^2 \to \mathbb{Z}_2$. Each subfigure illustrates a pair $(x, \vee(x))$ from this function: using an input data packet that corresponds to $x$, $A$ generates a data packet that corresponds to $\vee(x)$.

| $A$ | $B$ | $A \wedge B$ | $A \barwedge B$ | $(A \barwedge B) \barwedge (A \barwedge B)$ |
|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 0 | 0 | 1 | 0 |

**Table 5.4:** A truth table showing how Logical AND may be expressed solely in terms of NAND operations.

#### A Binary Operation: Logical AND

Our fourth example is Logical AND $\wedge : [\mathbb{Z}_2]^2 \to \mathbb{Z}_2$. The first three columns in Table 5.4 define Logical AND. The final column of Table 5.4 demonstrate how Logical AND can be achieved solely with NAND operations. From this we construct an A-type that represents Logical AND, which we illustrate in Figure 5.6.

#### A Ternary Operation

Our fifth example is the ternary operation that maps $[\mathbb{Z}_2]^3$ to $\mathbb{Z}_2$ and can be expressed as $(A \vee B) \wedge (B \wedge C)$. In the previous two examples we expressed both Inclusive-OR and logical AND solely in terms of NAND operations. By substituting these expressions into $(A \vee B) \wedge (B \wedge C)$ we generate the—rather verbose—expression given in Equation 5.1.

$$(((A \barwedge A) \barwedge (B \barwedge B)) \barwedge ((B \barwedge C) \barwedge (B \barwedge C))) \barwedge (((A \barwedge A) \barwedge (B \barwedge B)) \barwedge ((B \barwedge C) \barwedge (B \barwedge C))) \quad (5.1)$$

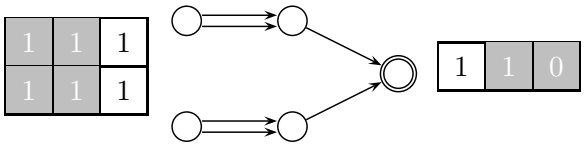
Table 5.5 is a truth table for $(A \vee B) \wedge (B \wedge C)$ and Table 5.6 is a truth table that, in comparison with Table 5.5, shows that Expression 5.1 is equivalent to $(A \vee B) \wedge (B \wedge C)$. From Expression 5.1 we construct an A-type that represents $(A \vee B) \wedge (B \wedge C)$, which we illustrate in Figure 5.7.

### 5.3.1 Existence

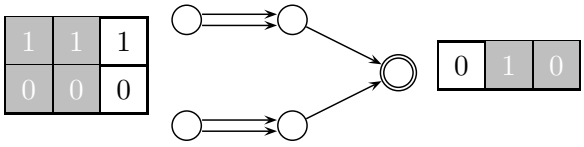
The examples in the previous section demonstrate how an A-type generates a data packet and how an A-type may represent a Boolean function. In fact, given any Boolean function there exists an A-type that represents that function. We formally present this next.

**Claim 5.1.** For any Boolean function $f : [\mathbb{Z}_2]^m \to [\mathbb{Z}_2]^n$ there exists a feedforward A-type without delay machines that represents $f$.

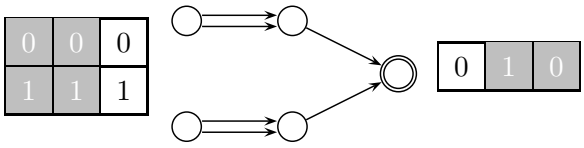
□



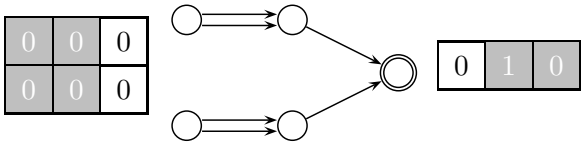

(a) $1 \wedge 1 = 1$




(b) $1 \wedge 0 = 0$




(c) $0 \wedge 1 = 0$




(d) $0 \wedge 0 = 0$

**Figure 5.6:** An A-type $A$, with delay $\delta = 2$, that represents Logical AND $\wedge : [\mathbb{Z}_2]^2 \to \mathbb{Z}_2$. Each subfigure illustrates a pair $(x, \wedge(x))$ from this function: using an input data packet that corresponds to $x$, $A$ generates a data packet that corresponds to $\wedge(x)$.

| A | B | C | $A \vee B$ | $B \wedge C$ | $(A \vee B) \wedge (B \wedge C)$ |
|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 | 1 |
| 0 | 1 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 |

**Table 5.5:** A truth table for the ternary operation $(A \vee B) \wedge (B \wedge C)$.

| A | B | C | $(A \barwedge A)$ | $(B \barwedge B)$ | $(B \barwedge C)$ | $(A \barwedge A) \barwedge (B \barwedge B)$ | $(B \barwedge C) \barwedge (B \barwedge C)$ | † | ‡ |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 |
| 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 |
| 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |
| 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 0 |

**Table 5.6:** A truth table showing that $(A \vee B) \wedge (B \wedge C)$ can be expressed using only NAND operators. † Note that the second rightmost column is given by the expression $((A \barwedge A) \barwedge (B \barwedge B)) \barwedge ((B \barwedge C) \barwedge (B \barwedge C))$. ‡ Also note that the rightmost column is given by expression 5.1.

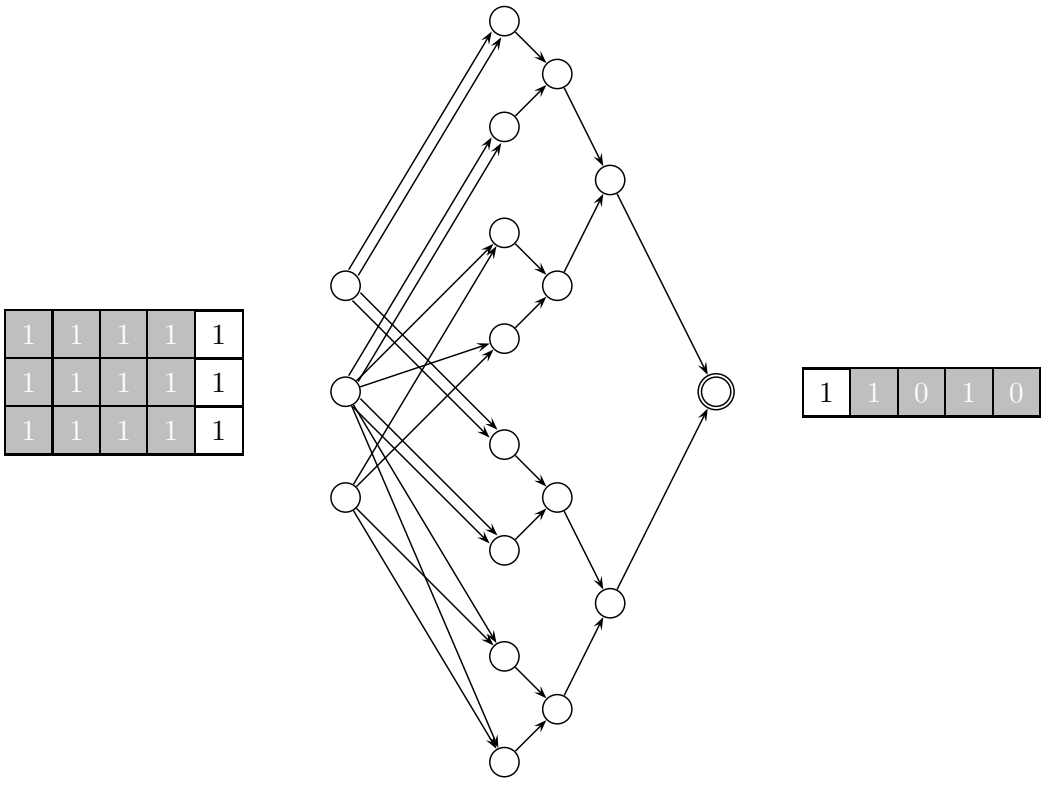



**Figure 5.7:** An A-type $A$, with delay $\delta = 4$, that represents the Boolean function $f : [\mathbb{Z}_2]^3 \to \mathbb{Z}_2$ given by $(A \vee B) \wedge (B \wedge C)$. This figure illustrates the pair $((1,1,1),1)$ from $f$. With an input data packet that corresponds to $(1,1,1)$, $A$ generates a data packet that corresponds to $f(1,1,1) = 1$. One can verify that $A$ represents $f$ by determining all eight $(x, f(x))$ pairs from $A$.

*Proof outline*

For any Boolean function $f : [\mathbb{Z}_2]^m \to [\mathbb{Z}_2]^n$ we can construct the following set of $n$ Boolean functions. $\{f_i : [\mathbb{Z}_2]^m \to \mathbb{Z}_2 \mid i \in [1, n]$, and for all $x \in [\mathbb{Z}_2]^m \ f_i(x)$ equals the $i$th entry of $f(x)\}$. We proceed to show that for each $f_i$ we can construct an A-type $A_i$ that represents $f_i$ and that these $A_i$'s can be assigned $m$ common input vertices to construct an A-type that represents $f$. Because NAND is expressively complete [76, p386] $f_i$ can be expressed as a bracketed expression with NAND operators. As we illustrated in Section 3.4.2, a bracketed expression can be represented by a binary tree: the leaves represent the arguments ($x_i$), the non-leaf nodes represent binary operations, and the topology of the graph represents the bracketing. So, each function $f_i$ can be represented with a binary tree. We convert this tree into an A-type directed graph $G_i$ as follows. Traverse $G_i$ in the following way. Start at the root of $G_i$ and then traverse $G_i$ so that all vertices are visited. Convert each edge into an arrow such that it is in the opposite direction to the first path made along that edge in the above traversal. The leaves of $G_i$ are its input nodes and the root of $G_i$ is its output node. Let us construct another A-type directed graph $G$ from $\{G_i\}$ by gluing together all leaves that have identical labels. Let $\delta$ denote the longest path in $G$. Finally, we construct an A-type with graph $G$ and delay $\delta$. This A-type is feedforward without delay machines that represents $f$, as required.

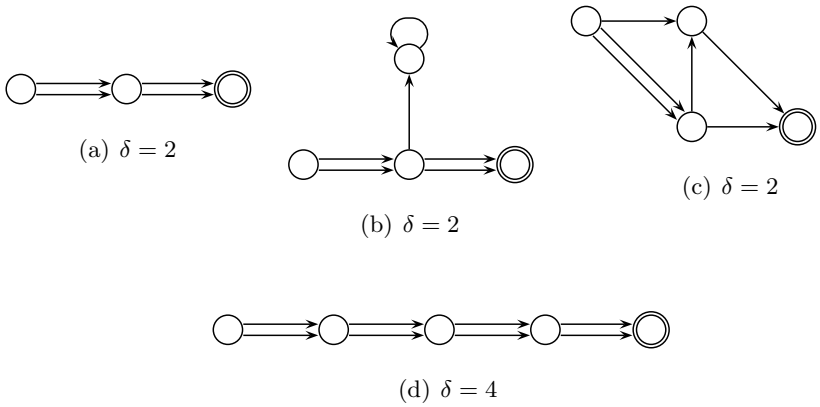
### 5.3.2 Uniqueness

In this section we prove that given a Boolean function there are always (infinitely) many A-types that represent that function.

In Figure 5.8 we show four A-types that represent the identity $id : \mathbb{Z}_2 \to \mathbb{Z}_2$. Figure 5.8(a) shows the identity A-type that we presented in the previous section. Figure 5.8(b) can be modified to give Figure 5.8(a) by removing the internal node that has no out-going arrows. We think of this node, and the arrows entering this node, as 'junk' because it has no affect on the output. The A-type shown in Figure 5.8(c) seems surprising in that there are paths of differing lengths going from the input node to the output node. In spite of our initial suspicion that these paths should be of equal length, the A-type shown represents identity. The A-type shown in Figure 5.8(d) can be constructed from two copies of the A-type shown in Figure 5.8(a), by letting the output node of one copy become the input node of the other copy. We can generalize this process of appending identity A-types to the output nodes of some A-type to construct another A-type (that has a longer delay) so that both A-types represent the same function. Consequently, there are infinitely many A-types that represent any given Boolean function.

When we search for A-types that represents a particular function we favour solutions with fewer nodes. Let us reconsider finding an A-type representation of the ternary expression $(A \vee B) \wedge (B \wedge C)$. Table 5.5 shows that the expressions $(A \vee B) \wedge (B \wedge C)$ and $B \wedge C$ are

equivalent. Since we have the A-type for Logical AND, we can quickly find another A-type to represent our ternary expression, as shown in Figure 5.9. There are several points to note about this alternative representation. First, we have A-types that succinctly represent $A \vee B$ and $B \wedge C$ but combining these in a manner suggested by $(A \vee B) \wedge (B \wedge C)$ does not provide the solution with the fewest vertices—perhaps those who are familiar with algebraic manipulation of Boolean expressions are not surprised by this result. Second, not only does this alternative A-type have fewer vertices but it also has a smaller delay. Third, we have a useful A-type that has one of its input nodes disconnected from the output nodes. When we implement evolutionary algorithms on populations of A-types the above points prove to be important considerations.



(a) $\delta = 2$

(b) $\delta = 2$

(c) $\delta = 2$

(d) $\delta = 4$

**Figure 5.8:** Four A-types that represent identity.

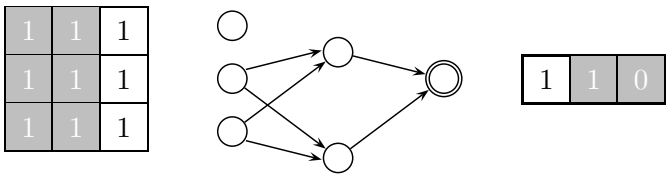



**Figure 5.9:** An A-type $A$, with delay $\delta = 2$, that represents the Boolean function $f : [\mathbb{Z}_2]^3 \to \mathbb{Z}_2$ given by $B \wedge C$. The A-type shown in Figure 5.7 and $A$ represent the same functions. Here we demonstrate $A$ generating a data packet that corresponds to $f(1, 1, 1) = 1$.

### 5.3.3 Sequential Input

The examples in the previous section illustrate A-types with clamped (non-sequential) input. Let us reconsider the feedforward A-types that we construct to represent Boolean functions

(this construction was presented in our proof for Claim 5.1). A Boolean function maps a single column of Boolean values to another single column of Boolean values. So if an A-type $A$ represents a Boolean function then it generates a single column data packet given another single column data packet. The 'shunt bits' are columns that are identical to the input data packet, so $A$'s input bits are fixed. Because $A$'s delay equals the length of $A$'s longest path and because $A$ is feedforward, at successive moments if $A$'s input nodes are fixed then $A$'s output will be identical. That is, $A$ is operating in a clamped (non-sequential) fashion. As we noted in Section 3.5 ANNs with a clamped input are a special case of ANNs with sequential input. We designed Definition 5.1 so that we could easily implement A-types with clamped inputs using A-types with sequential input—as demonstrated above.

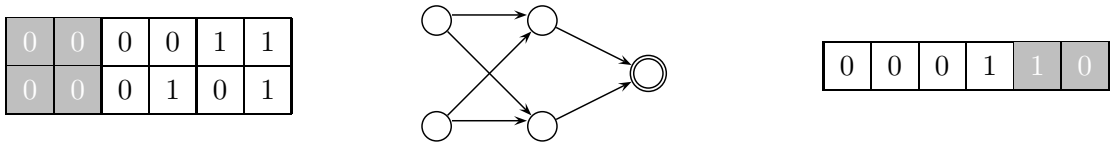



**Figure 5.10:** An A-type $A$ that represents columnwise Logical AND of length 4, $\wedge_4$. Here we illustrate $A$ accepting and generating data packets that correspond to ( $((1,1)(1,0)(0,1)(0,0))$, $\wedge_4$ $(1,0,0,0)$ ).

Now we demonstrate A-types with sequential input. Let us consider colunmwise Boolean functions (defined in Section 2.3). For instance, let us consider columnwise Logical AND of length four, $\wedge_4 : M[4,2] \rightarrow M[4,1]$. This function has $2^8 = 256$ $(x, \wedge_4(x))$ pairs. In Figure 5.10 we illustrate an A-type that represents $\wedge_4$ and the data packet it generates that corresponds to one such $(x, \wedge_4(x))$ pair. This A-type is identical to that shown in Figure 5.6. More generally, this A-type represents columnwise Logical AND of length $l$ for all positive, finite length integers $l$. Furthermore, we have similar results for the other four A-types that represent Boolean functions described in this section. That is, each of these A-types also represent the corresponding columnwise Boolean function for all finite, positive integer lengths. These examples motivate the following claim.

**Claim 5.2.** For any columnwise Boolean function $f$ there exists a feedforward A-type that represents $f$.

□

If the reader compares Claim 5.1 with Claim 5.2 then they may note that in Claim 5.2 we no longer exclude A-types that have delay machines. Now we argue that delay machines are necessary for our A-types to process sequential input.

### 5.3.4 The Necessity of Delay Machines

If the reader re-examines Turing's definition of an A-type unorganised machine, which we reproduced in Section 3.7.1, then it is apparent that Turing did not specify a means of synchronizing information flow through the network. We make the following claim.

**Claim 5.3** (Necessity of Delay Machines)**.** There exist columnwise Boolean functions that can not be represented by A-types without delay machines.

□

We support this claim by providing evidence that columnwise Exclusive-OR cannot be represented by an A-type without delay machines.

#### Supporting Evidence: Desynchronized Path Lengths

First, we construct an A-type that represents Exclusive-OR, mindful that it may also represent columnwise Exclusive-OR. Let us devise an A-type that represents Exclusive-OR $\oplus : [\mathbb{Z}_2]^2 \rightarrow \mathbb{Z}_2$. We make use of A-types that we constructed in previous sections and join them to give an A-type that represents $\oplus$. In particular, we use the equality $A \oplus B = (A \vee B) \wedge (A \barwedge B)$ (proven in Figure 5.11(a)) to construct an A-type that represents $\oplus$ from an A-type that represents $\vee$ and an A-type that represents $\wedge$. This constructed A-type is shown in Figure 5.11(b); it represents $\oplus$, but we can easily verify (by processing data) that it does not represent columnwise Exclusive-OR. This is because the time (number of moments) required to represent $A \vee B$ differs from the time required to represent $A \barwedge B$—the information percolating through the A-type becomes desynchronized. Figure 5.11(b) shows that our A-type for $A \vee B$ has a delay of two and our A-type for $A \barwedge B$ has a delay of unity. Because we have to combine (with AND) the outputs of these two A-types we have to introduce a further delay of one moment to the output of the A-type that performs $(A \vee B)$. To this end we introduce a delay machine, the resulting A-type is shown in Figure 5.11(c). Note that all of the A-types that we devised that represent identity (for sequential input) have an even integer delay. These examples suggest* that we cannot employ A-types (without delay machines) that represent identity to repair the A-type shown in Figure 5.11(b)—we return to this point at the end of this section.

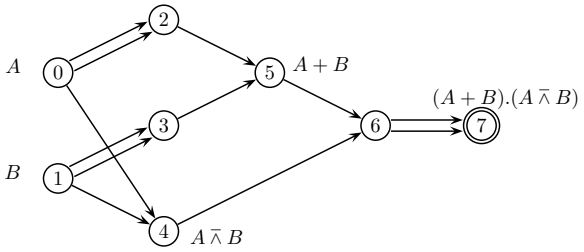
#### Supporting Evidence: Searching for Counter-Examples

Second, we run computer simulations that search for a counter example to Claim 5.3. In particular, we conduct searches for A-types that represent columnwise Exclusive-OR. We

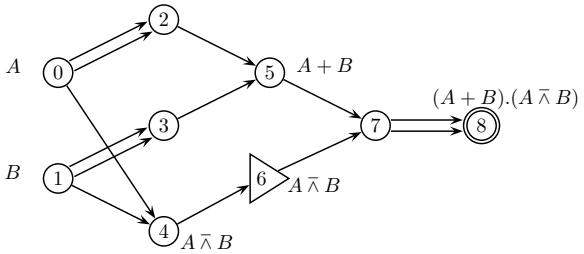
*It was an experimental investigation, via computer simulation, that motivated Claim 5.3. We outline this investigation in the next section and present its results in Chapter 7. These experimental results provide compelling evidence that Claim 5.1 is true.

| $A$ | $B$ | $A \wedge B$ | $A \vee B$ | $A \barwedge B$ | $(A \vee B).(A \barwedge B)$ | $A \oplus B$ |
|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 | 1 | 1 |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 |

(a) A truth table proving that $(A \vee B).(A \barwedge B) = A \oplus B$.




(b) Composing an A-type to represent $(A \vee B).(A \barwedge B) = A \oplus B$. Note that the subgraph generated by the node set $\{0, 1, 2, 3, 5\}$ represents $(A \vee B)$ (see Figure 5.5). Also, the subgraph generated by the node set $\{0, 1, 4\}$ represents $A \barwedge B$. Furthermore, the subgraph generated by the node set $\{4, 5, 6, 7\}$ represents AND.




(c) Inserting a delay machine into the A-type shown in 5.11(b). This ensures that the two inputs into node 7 are synchronized.

**Figure 5.11:** Using an expression that involves AND, NAND, and Inclusive-OR to generate an A-type that represents Exclusive-OR.

conduct two such searches: one using A-types without delay machines, the other using A-types with delay machines. We present this investigation in Chapter 7. This investigation results in many A-types that represent columnwise Exclusive-OR with all containing delay machines. This is compelling evidence that Claim 5.3 is true.

#### Supporting Evidence: A Simpler Hypothesis

Third, we present a claim that is a useful alternative to Claim 5.3. This claim is easier to test experimentally.

When we discussed information percolating through the A-type shown in Figure 5.11(b) we noted that inserting A-types without delay machines that represented identity would not remedy the problem of desynchronised data. From this observation we are lead to make the following two claims.

**Claim 5.4.** There is no A-type without delay machines and with an odd delay that represents columnwise identity $id : M[l, 1] \rightarrow M[l, 1]$ for some $l \in \{2, 3, 4, \ldots\}$.

□

**Claim 5.5** (A Simpler Hypothesis)**.** Claim 5.3 implies Claim 5.4.

□

*Outline of proof for Claim 5.5* We show that if Claim 5.3 is false then so to is Claim 5.4. Consider a columnwise Boolean function $f$. We can construct a (feedforward) A-type $A$ without delay machines that represents the (non-columnwise) Boolean function $f$ (see Claim 5.1). If Claim 5.4 is false then we can construct A-type machines that can be inserted into $A$ to ensure that all calculations are synchronized. This synchronized A-type represents $f$ showing that Claim 5.3 is false.

We ran computer simulations to search for a counter-example to Claim 5.4. In particular, we searched for A-types without delay machines that represent columnwise identity $id : M[l, 1] \rightarrow M[l, 1]$, $l \in \{2, 3, \ldots\}$. We examined the delay of each solution to this search. We present this investigation in Chapter 7. This investigation gave many A-types that represent $id$ none of which had an odd delay. This provides further evidence that Claim 5.3 is correct.

### 5.3.5 Clamped is Easier than Sequential

When we employ our A-types with clamped inputs and clamped outputs to represent Boolean functions we do not require delay machines. Furthermore, when using A-types in this manner every input data packet consists of rows of identical bits. Whereas, when using A-types with sequential input, in general, the bits in a given row of an input data packet differ. If we consider training examples of some fixed length $l$ then the set of all clamped training examples of some concept $c$ is a proper subset of the set of all sequential examples of $c$. Consequently,

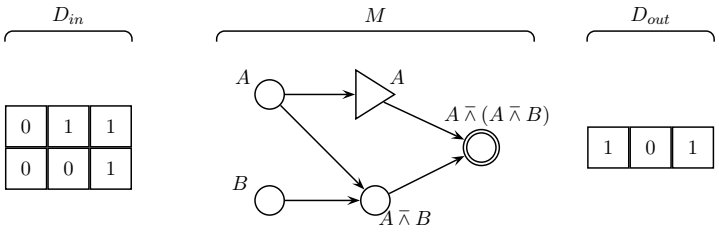
for a given problem, finding the serial solution is more difficult than finding the clamped solution. Furthermore, an A-type that represents some columnwise Boolean function $f$ also represents $f$ in the clamped scheme.

### 5.3.6 Alternative Input Schemes

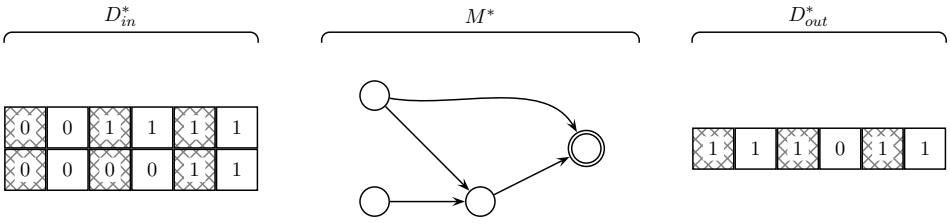
Teuscher [2, p31] presents a definition of an A-type machine that has input and output nodes. This interpretation of Turing's A-types only requires nand machines; however, these networks require particular attention to their timing [2, p67]. For these networks to process columnwise input they require two clock speeds and the ratio of these two speeds is $\delta$. We interpret these networks as A-types without delay machines that can represent columnwise Boolean functions where the input and output data packets are processed in a particular way. We outline this with an example that is illustrated in Figure 5.12. The A-type $M$ shown in Figure 5.12(a), with a delay $\delta = 2$, represents columnwise $A \barwedge (A \barwedge B)$. In this figure we show $M$ with an input-output pair $(D_{in}, D_{out})$. In Figure 5.12(b) we show a second A-type $M^*$ with an input-output pair $(D^*_{in}, D^*_{out})$. We can transform $D_{in}$ to $D^*_{in}$ by making two (because $\delta = 2$) copies of every column in $D_{in}$. Also, we can transform $D^*_{out}$ to $D_{out}$ by only retaining every second (because $\delta = 2$) entry of $D^*_{out}$. With this procedure we could define how $M^*$ represents columnwise $A \barwedge (A \barwedge B)$. Furthermore, we could generalize this procedure to allow A-types without delay machines to represent columnwise Boolean functions. However, such a definition must include pre-processing of input data packets and post-processing of output data packets, as suggested by the above example.

### 5.3.7 Beyond Boolean Functions

We have invested substantial effort ensuring that our A-types can function as sequentially driven dynamic ANNs. However, the only sequential examples that we have provided, so far, in this chapter are columnwise Boolean functions. Furthermore, all A-types illustrated, so far, in this chapter are feedforward. Our A-types are more general than those examples may suggest. For instance, consider the A-type shown in Figure 5.13. There is no columnwise Boolean function that is represented by this A-type. Furthermore, clamped Boolean functions are a particularly special class of columnwise Boolean functions. For every input-out pair of such a function the $i$th column of the output is entirely determined by the $i$th column of the input. In general, this property does not hold for a sequential function. For example, consider sequential addition: the $i$th output column is determined by the $i$th input column and 'carry' information from the $(i-1)$th output column. In principle, we can construct an A-type that represents serial Boolean addition for arbitrarily long input and output data packets. In chapter 7 we devise a simple sequential function that requires 'carry' information (that is each output column is determined by more than one input column). We use this function as one of the benchmark concepts searched for when we compare the performance




(a) An A-type $M$, with a delay $\delta = 2$, that represents columnwise $A \barwedge (A \barwedge B)$ and a particular input-output pair from this Boolean function.




(b) An A-type $M^*$ without delay machines, which given $D^*_{in}$ generates $D^*_{out}$. By deleting the cross-hatched columns in $D^*_{in}$ and $D^*_{out}$ we could define $M^*$ to also represent columnwise $A \barwedge (A \barwedge B)$.

**Figure 5.12:** An A-type $M$ that represents a function $f$ and a similar A-type $M^*$ without delay machines. The input and output data packets may be processed so that we can consider $M^*$ to also represent $f$.

of our A-type EAs. Finally, let us touch upon the limitations of our A-types. Because we defined our A-types as finite state machines we can consult the literature to understand the limitations of our A-types. Given infinite sequence of input columns an A-type with a finite number of nodes cannot represent functions on this input that require an 'infinitely long carry' such as sequential multiplication. To make this precise we quote the following theorem from Minsky [33, p27]

> No fixed finite state machine can multiply arbitrarily large pairs of binary (or decimal) numbers.

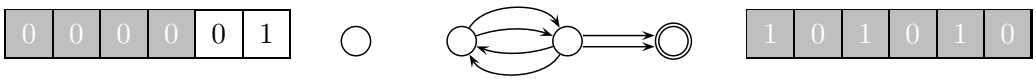



**Figure 5.13:** An A-type with oscillatory output that is independent of the input.

# 6 A Possible 'Genetical Search'

## 6.1 Aims of this Chapter

Here we detail an EA whose population is a set of A-types. We call this algorithm *genetic_search_one*. We devised this algorithm and encoded it into a computer program—an outline of which is given in Appendix A. In this chapter we present the following: an overview of *genetic_search_one*; the reduction of this algorithm into two special cases, namely a blind search and an EA without crossover; and details of each special case.

Note that we present the results of our experimental tests of *genetic_search_one* in Chapter 7.

In Chapter 3 we introduced EAs and we concluded that it is likely that these algorithms were what Turing intended when he wrote of a 'Genetical Search'. In this chapter we apply EAs to our interpretation of Turing's A-type unorgansied machines.

Teuscher used linear chromosome representations of B-types when he implemented EAs on Turing networks. Teuscher [2, p88] used B-types with lists that prescribed whether each connection modifier was in a 'connected' or 'disconnected' state. These lists give linear chromosomes for Teuscher's A-types. We employ graph chromosomes to enable us to be particular about our genetic operators. This allows us to employ graph theoretical ideas, such as connectedness, in an effort to construct a useful crossover operator. It is with these graph theoretic ideas that we attempt to construct a useful crossover algorithm. Because A-types are relatively simple artificial neural networks our task of encoding particular graph chromosomes is achievable.

## 6.2 Supervised Learning with A-Types

Many EAs employ supervised learning [43, p28]. We too employ supervised learning in *genetic_search_one*. In this section we describe how supervised learning can be applied to A-types.

### 6.2.1 Sets of Training Examples

Recall from Section 3.3 that supervised learning uses a set of training examples; each training example is a pair of input and expected output. These pairs are used to train candidate solutions. When we train A-types on a particular problem, all candidate solutions have the

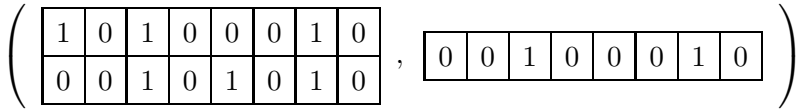



**Figure 6.1:** A training example. In this case we illustrate a pair from columnwise Logical AND of length eight.

same input dimension; similarly, all candidate solutions have the same output dimension. So, when we use supervised learning with A-types each training example is a pair of data packets. All of the input packets have the same number of rows, namely, the input dimension of every candidate solution. Similarly, all expected output data packets have the same number of rows, namely, the output dimension of every candidate solution. Consider a training example $E$ whose input data packet is $D_{in}$ and whose expected output data packet is $D_{ex}$. We assess the performance of an A-type candidate solution $M$ with respect to $E$ by first using $M$ with input $D_{in}$ to compute an output data packet $D_{out}$, and then comparing $D_{out}$ with $D_{ex}$. To assess $M$'s performance with respect to all training examples we average $M$'s performance with respect to each training example. We detail our assessment of an A-type's performance and our method of averaging over several training examples in Section 6.3.3—there we detail *genetic_search_one*. In Figure 6.1 we illustrate a set of training data that is suitable for supervised learning with A-types. This set has a single example of columnwise Logical AND.

### 6.2.2 Particulars of our Implementation

When we search for an A-type $A$ that represents a given concept we have to search for both $A$'s graph and $A$'s delay $\delta$. In all of the algorithms described in this chapter we choose an A-type graph, estimate a range of possible delays for that graph, and determine the fitness of each (graph + delay) A-type. Rather than always constructing a single A-type, we construct an A-type graph and consider the A-types that result from a range of delays because it is easy to code and efficient to run.

In *genetic_search_one* (and in its two special cases) we construct an A-type graph $G$ and consider all A-types with $G$ and a delay that is an element of an interval $[\delta_{min}, \delta_{max}]$, we illustrate this in Figure 6.2. When we conduct supervised learning for each training example $E$ all $(\delta_{min} - \delta_{max} + 1)$ A-types are trained with respect to $E$. We do this by elongating $E$'s input data packet and using the A-type $(G, \delta_{min})$ to compute a similarly elongated output data packet. Sections of this output data packet are then compared to the expected output of $E$. We outline this algorithm in Table 6.1 and provide an illustrative example in Figure 6.3.

Because we encode an A-type graph and a range of delays an individual in our algorithms is actually a set of A-types all of which have the same A-type graph. So in our algorithm descriptions when we say that we make an A-type we are actually making a set of A-types.

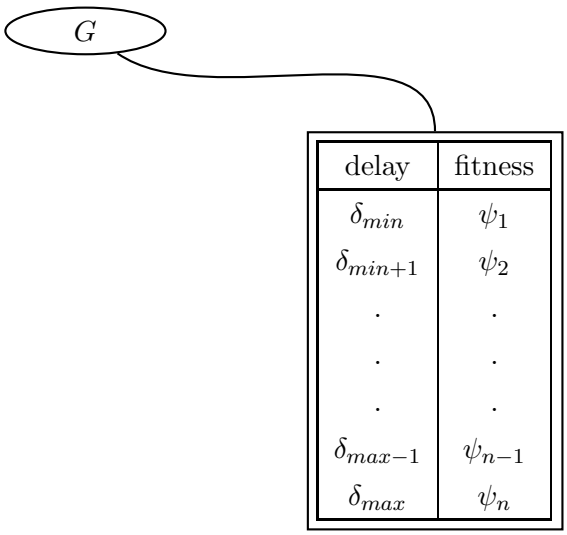



**Figure 6.2:** An illustration of how we record the performance with respect to a set of training examples of a set of A-types that have the same graph $G$.

**Training A-types That Have Identical Graphs**

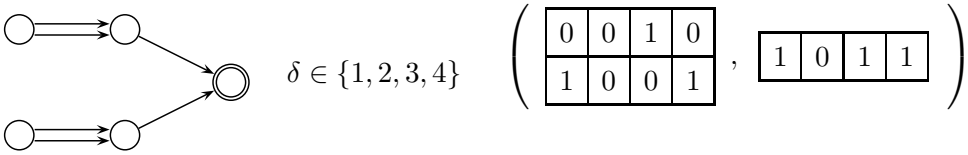
Given an A-type graph $G$, an estimate of its minimum delay $\delta_{min}$ and an estimate of its maximum delay $\delta_{max}$. Furthermore, given a training example $(D_{in}, D_{ex})$, where $D_{in}$ is a $m \times p$ data packet and $D_{ex}$ is a $n \times q$ data packet.

1. The input data packet $D_{in}$ is elongated by copying the leftmost column of bits to produce a $1 \times m$ vector and appending it to the left end of $D_{in}$. This process is repeated $(\delta_{max} - \delta_{min})$ times. We shall denote this elongated data packet as $D'_{in}$.

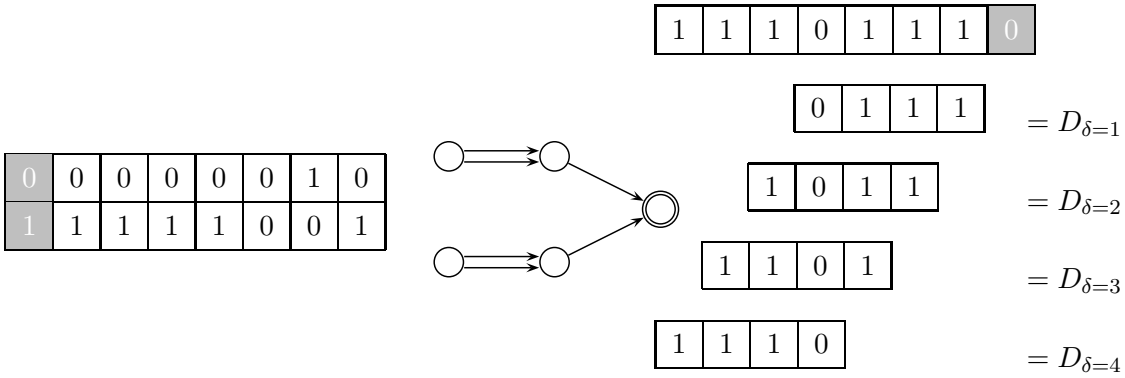
2. Use the A-type that has the graph $G$ and a delay $\delta_{min}$ to compute the output data packet $D'_{out}$.

3. Consider each data packet of length $p$ that is constructed by adding $p$ consecutive columns of $D'_{out}$. Compare each such data packet with $D_{ex}$.

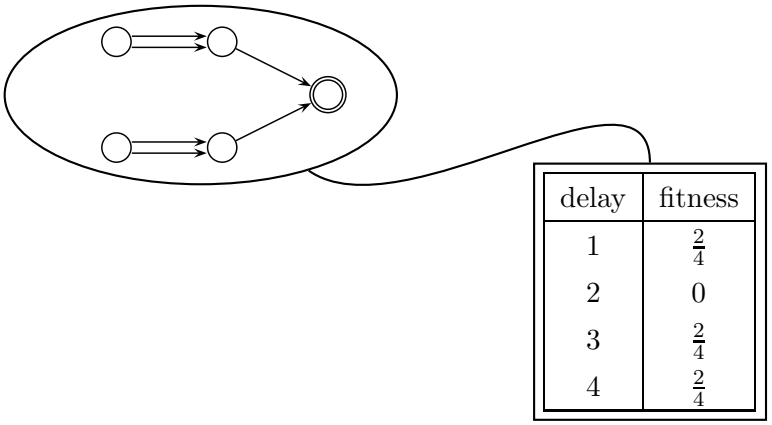
**Table 6.1:** Conducting supervised learning on a set $S$ of A-types that all have the same graph and whose delays are members of an interval $[\delta_{max}, \delta_{min}]$. The above outline details how we train all A-types in $S$ on a particular training example.




(a) Considering an A-type graph $G$, a range of delays, and a training example $(D_{in}, D_{ex})$.




(b) Elongating $D_{in}$ and and using this elongated data packet $D'_{in}$ as input to $(G, \delta = 1)$ to generate a long data packet $D'_{out}$ to determine the output generated by each of the four A-types.



| delay | fitness |
|---|---|
| 1 | $\frac{2}{4}$ |
| 2 | 0 |
| 3 | $\frac{2}{4}$ |
| 4 | $\frac{2}{4}$ |

(c) An individual in our EA is an A-type graph and a list of (delay, fitness) pairs. In this case the individual represents four A-types. Note that in this diagram we display the fitness as the Hamming distance between the expected output $D_{ex}$ and the actual output $D_{out}$.

**Figure 6.3:** Using the input data packet of a training example to generate the output from four A-types that have the same graph.

### 6.2.3 Estimating Minimum and Maximum Delays

When we construct an individual $A$ in our algorithms (either a randomly constructed member of the initial population, or the result of crossover or mutation) we need to estimate a suitable range of delays for $A$. If $A$ has $N$ nodes then $A$'s delay is less than $2^N$. The larger the range of delays for each individual the longer it takes to train each individual. We take a somewhat pragmatic approach to estimate the range of delays. To estimate the minimum delay we perform the following four steps. First, we collect $A$'s output $D_{out}$ from sequentially entering a long data packet with random entries. Second, we collect $A$'s output $D'_{out}$ from sequentially entering another long data packet with random entries. Third we determine the position $l$ where $D_{out}$ and $D'_{out}$ first differ (if $D_{out} = D'_{out}$ then we set $l = -1$). Four, we subtract the sum of $A$'s input dimension and $A$'s output dimension from $l$. If $l$ is negative then we set it to zero. Our estimate of $A$'s minimum delay is $l$. Our estimate of $A$'s maximum delay is the number of nodes in $A$. Our estimate of an appropriate range of delays is not guaranteed to be optimal.

## 6.3 Evolutionary Algorithms Applied to A-Types

Here we detail *genetic_search_one* and two special cases of this EA.

### 6.3.1 An Outline of the General Case

The algorithm *genetic_search_one* is a steady-state EA with a population of A-types. We give an outline of the algorithm in Table 6.2. Note that although this algorithm is detailed in Section 6.3.3 we present an overview here. This is necessary for the next section where we discuss the two special cases of *genetic_search_one*.

### 6.3.2 Three Incarnations

Our algorithm *genetic_search_one* has many parameters some of which can be set to zero. Consequently, we can devise algorithms that are special cases. We devise and implement two such special case algorithms. This gives us three algorithms that we can compare. These algorithms are summarized below.

1. Blind search: We can trivialize *genetic_search_one* so that each generation simply involves the creation of a single random A-type and a test of that A-type's performance with respect to the training set. Each unsuccessful A-type is destroyed and the creation of the next A-type is not affected by the performance of any of the previous attempts. In biological terms we say that for such a search no individual exhibits inheritance. Consequently, we do not class the search as a EA in spite of it being a special case of one. We call this search *blind_search_one*.

*genetic_search_one*

1. Create an initial population. That is, create a number of A-types, such that each A-type is randomly generated, and store these A-types in a multiset that we call the population.

2. Repeat until either the population contains a fit enough A-type or a maximum number of attempts have been performed. The number of these outermost iterations is recorded as the generation.

   a) Repeat a set number of times. That is, perform a set number of crossovers.

      i. We invoke a breeder selection rule twice on the population to reference two A-types that will be parents for a crossover operation.
      ii. With the two parent A-types we perform crossover and we call the result the child A-type.
      iii. We train the child and put it into the population.
      iv. We invoke a termination rule on the population to select an element of the population. This element is deleted from the population.

   b) Repeat a set number of times. That is, perform a set number of mutations.

      i. We invoke a mutant selection rule on the population to reference an A-type that will be the original for a mutant.
      ii. With the original A-type we perform a mutation and we call the result the mutant A-type.
      iii. We train the mutant and put it into the population.
      iv. We invoke a termination rule on the population to select an element of the population. This element is deleted from the population.

3. Return the fittest A-type in the population. If there is more than one A-type with the population's lowest fitness then we randomly select an element from the set of such individuals.

**Table 6.2:** A outline of *genetic_search_one*. Note that we formally present this algorithm in Section 6.3.3. Also, in later sections of this chapter we present detail about this algorithm. For instance, details of the fitness function, the mutation function, and the crossover function are given.

2. EA without crossover: At first glance it may seem that all evolutionary searches require the exchange of genetic material between individuals to create offspring. This is not so*, there are EAs that do not have a crossover operator. The second algorithm that we use is a special case of *genetic_search_one* that has no crossover operators. We call this search *mutation_search_one*.

3. EA with crossover: *genetic_search_one*.

Table 6.3 illustrates how we classify these three algorithms.

| | *blind_search_one* | *mutation_search_one* | *genetic_search_one* |
|---|---|---|---|
| population size | 1 | $> 1$ | $> 1$ |
| has mutation operator | yes | yes | yes |
| has crossover operator | no | no | yes |

**Table 6.3:** Classifying our three algorithms.

For the remainder of this chapter we present these three algorithms. First, we present *genetic_search_one*. Second, we present *blind_search_one*; in this description we detail the fitness function that we use in all three algorithms. Third, we present *mutation_search_one*; in this description we detail the mutation function, which we also employ in *genetic_search_one*. Finally, we detail *genetic_search_one*'s crossover operator.

### 6.3.3 Detailing the General Case

Here we detail *genetic_search_one* (recall that we presented an outline of this algorithm in Table 6.2). We present this via the main listing in Table 6.4 and its two subroutines in Table 6.5 and Table 6.6. We divide our presentation into a main routine and two subroutines to clarify the description. The first subroutine, *initial_pop*, details how we generate the initial population. The second subroutine, *evolve*, details how the population evolves.

The generation counter is not particularly significant in *genetic_search_one*. A more important statistic, with respect to the control of the algorithm, is the total number of births (assigned to the variable `births` in *evolve_one*). The total number of births is more useful than the number of generations when we compare *genetic_search_one* to a blind search because each generation can denote many steps; whereas, `births` denotes the number of A-types created.

It is possible that *genetic_search_one* fails to find a solution. The algorithm terminates if either a solution is found or a specified number of candidate solutions have been constructed. The parameter $N_{births}$ is a lower bound of the maximum number of A-types that can be created. For example consider the case where $N_{cross} = 34$, $N_{muts} = 33$, and the search is

*In Section 6.5 we elaborate on the fact that crossover is not necessary for evolution.

**genetic_search_one**$(n, p, d, l, u, N_{births}, f_{worst}, N_{cross}, N_{muts})$ is the EA that is outlined in Table 6.2. This description uses two subroutines. The first subroutine, *initial_pop*, generates the initial population and the second subroutine, *evolve*, evolves the population until either a solution is found or a maximum number of attempts have been performed. Both subroutines are detailed later in this section.

| Initial Population Parameters | | |
|---|---|---|
| *Type* | *Parameter* | *Description* |
| $\mathbb{Z}^+$ | $N_{init}$ | Number of initial A-types. |
| $\mathbb{Z}^+$ | $m$ | Input dimension of all A-types. |
| $\mathbb{Z}^+$ | $p$ | Output dimension of all A-types. |
| $[0, 1]$ | $d$ | Probability that a node is constructed as a delay machine. |
| $\mathbb{Z}^+$ | $l$ | Lower bound of the number of nodes in any initial A-type. |
| $\mathbb{Z}^+$ | $u$ | Upper bound of the number of nodes in any initial A-type. |
| Evolution Parameters | | |
| $\mathbb{Z}^+$ | $N_{births}$ | Maximum number of times that an A-type will be constructed. |
| $[0, 1]$ | $f_{worst}$ | Worst (largest) fitness acceptable for a solution A-type. |
| $\mathbb{Z}^+$ | $N_{cross}$ | Number of crossovers performed between each generation. |
| $\mathbb{Z}^+$ | $N_{muts}$ | Number of mutations performed between each generation. |

The algorithm has the following steps

1. *Create an initial population:* We evoke the method that generates an initial population.
   $P \leftarrow initial\_pop(N_{init}, p, d, l, u)$
2. *Check for a solution:* We check the population for a solution because we may stumble on a solution when creating the initial population.
3. *Evolve the population:* We evoke the method that evolves the population. The output from this method may be a solution.
   `soln` $\leftarrow evolve(P, N_{births}, f_{worst}, N_{cross}, N_{muts})$
4. *Check solution:* The method *evolve* terminates if a solution is found or a specified number of attempts are exceeded. If the former is true then we return `soln`, otherwise the search fails.

**Table 6.4:** A description of *genetic_search_one*. Note that this listing calls the subroutines *initial_pop* and *evolve*.

**initial_pop**$(N, m, p, d, l, u)$ generates a population $P$ of A-types. Each A-type is restricted by the following constraints: there are specified input and output dimensions (both of which can be determined form the training set); there is an admissible range for the number of nodes in the A-type; and each node has a specified probability that it is constructed as a delay machine (otherwise it is a nand machine). The construction of each A-type is a random selection from the set of A-types that conform to the above constraints.

| Parameters | | |
|---|---|---|
| *Type* | *Parameter* | *Description* |
| $\mathbb{Z}^+$ | $N$ | Number of A-types in the returned population $P$. |
| $\mathbb{Z}^+$ | $m$ | Input dimension of all A-types in $P$. |
| $\mathbb{Z}^+$ | $p$ | Output dimension of all A-types in $P$. |
| $[0, 1]$ | $d$ | Probability that a node is constructed as a delay machine. |
| $\mathbb{Z}^+$ | $l$ | Lower bound of the number of nodes in any A-type in $P$. |
| $\mathbb{Z}^+$ | $u$ | Upper bound of the number of nodes in any A-type in $P$. |

The algorithm has the following steps

1. *Construct $N$ A-types:* Repeat $N$ times.
   a) *Choose the A-type's size:* `size` $\leftarrow$ a random integer between $l$ and $u$ inclusive.
   b) *Construct the A-type:* We construct an A-type $A$ with $m$ input nodes, $p$ output nodes and $(\texttt{size} - (m + p))$ internal nodes. Each internal node is constructed such that the probability that it is a delay node is $d$ and the probability that it is a nand node is $(1 - d)$. The arrows are inserted randomly between the nodes restricted by the condition that $A$ is a valid A-type.
2. Return $P$.

**Table 6.5:** A description of the subroutine *initial_pop*. This creates the initial population for *genetic_search_one*.

**evolve**$(P, N_{births}, f_{worst}, N_{cross}, N_{muts})$ takes a population and changes it over several generations by inserting new A-types and deleting existing A-types. Between each generation three operations are invoked: a set number of crossovers; a set number of mutations; and terminations such that the population size for each generation is constant.

Parameters

| *Type* | *Parameter* | *Description* |
|---|---|---|
| { A-types } | $P$ | population of A-types that has been generated by *initial_pop*. |
| $\mathbb{Z}^+$ | $N_{births}$ | Maximum number of A-type constructions. |
| $[0, 1]$ | $f_{worst}$ | Worst (largest) fitness acceptable for a solution A-type. |
| $\mathbb{Z}^+$ | $N_{cross}$ | Number of crossovers performed between each generation. |
| $\mathbb{Z}^+$ | $N_{muts}$ | Number of mutations performed between each generation. |

The algorithm has the following steps

1. *Update birth count:* `births` $\leftarrow |P|$
2. *Iterate through generations:* We call each iteration through this loop a generation. We exit if a set number of A-types are constructed or a solution is discovered.
   While($births \leq N_{births}$) do
   a) *Perform a number of crossovers:* Repeat $N_{cross}$ times.
      i. *Crossover:* We invoke the breeder selection rule twice to choose two parents. We then invoke the crossover rule to create a new A-type that we call `child`.
      ii. *Training:* We train `child` with respect to the training set $T$.
      iii. *Adding child to P:* Add `child` to $P$ and increment `births` by one.
      iv. *Exit if we have a solution:* If the fitness of `child` is less than $f_{worst}$ then we exit the current (crossover) loop and go to step 3.
      v. *Terminate an individual:* We invoke our termination rule to select an individual. This A-type is deleted from $P$.
   b) *Perform a number of mutations:* Repeat $N_{muts}$ times.
      i. *Mutate:* We invoke the mutate selection rule to choose an A-type. We then invoke the mutation rule to create a new A-type that we call `mutant`.
      ii. *Training:* We train `mutant` with respect to the training set $T$.
      iii. *Adding mutant to P:* Add `mutant` to $P$ and increment `births` by one.
      iv. *Exit if we have a solution:* If the fitness of `mutant` is less than $f_{worst}$ then we exit the current (mutation) loop and go to step 3.
      v. *Terminate an individual:* We invoke our termination rule to select an individual A-type. This A-type is deleted from $P$.
3. Return the fittest element of $P$.

**Table 6.6:** The subroutine *evolve*.

conducted but no solution is ever found. In that case the total number of births will be ($N_{births}$ + 34 +33). In most instances the total number of generations will be large and the upper bound allows us to terminate the search if is taking an 'unreasonably' long time. Since $N_{births}$ will be very large when we quote this statistic we do not add the few extra births to make it precise.

The mutant selection rule, the breeder selection rule, and the termination rule, give a great deal of freedom to this algorithm. If the analogy of biological evolution is followed closely then the appropriate specifications of these rules are as follows.

> *Mutant selection rule:* this should be random; that is, independent of the fitness of the individuals.
>
> *Breeder selection rule:* this should be based on the fitness of members of the population. In both selective breeding (as in the selective breeding of plants to produce high yielding crops) and natural selection the selection is fitness based[†].
>
> *Termination rule:* this should be based on the fitness of members of the population. This would be analogous to selected breeding or the analogy of natural selection. Although allowing some proportion of the terminations to be random may allow us to model death by old age and death by accident—consequently that may be a useful alternative.

In spite of (or perhaps because of) our suspicion that the three rules should be as specified above, we investigate the performance of the EA with various states of these rules. This can be seen in the results in Chapter 7. This investigation provides evidence whether the biological analogy is useful for determining the selection rules.

In *evolve* the termination rule is invoked shortly after each addition of an A-type to the population. So in *genetic_search_one*, after the initial population has been created, the population's size only ever fluctuates by one. That is, *genetic_search_one* is a steady-state EA.

## 6.4 A Blind Search

A simple blind search is a good test for our more sophisticated search algorithms. Our implementation of an EA applied to A-types required a substantial amount of computer programming. Consequently, it is sensible to compare our EAs to a scheme where A-types are randomly constructed and then tested on the training set until a solution is discovered. This random search is easily implemented. So, unless its performance is poor in comparison, it may be preferable to the EA as a concept learning strategy. In this section we detail *blind_search_one*.

[†]For selected breeding the fitness criteria is altered by humans.

*blind_search_one*

1. Repeat until either a fit enough A-type is constructed or a maximum number of attempts are performed.
   a) Randomly generate an A-type.
   b) Train this A-type.
2. Return the A-type that is fit enough to terminate this search.

**Table 6.7:** An outline of *blind_search_one*. Note that we detail this algorithm in Section 6.4.4.

### 6.4.1 An Outline of our Blind Search

We give an outline of *blind_search_one* in Table 6.7. Note that this algorithm is not detailed until Section 6.4.4, the current presentation is a brief overview to aid our discussion.

From the outline shown in Table 6.7 we see that *blind_search_one* simply randomly generates an A-type and terminates if this A-type is a solution; otherwise the A-type is destroyed and another attempt made. The search also terminates if the number of attempts exceeds a specified maximum.

Before we provide further detail of *blind_search_one* we explain how we randomly generate an A-type and detail its fitness function.

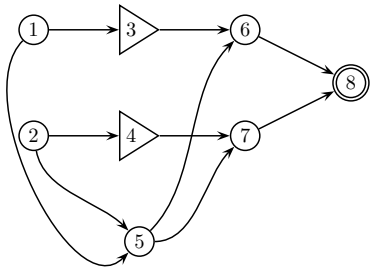
### 6.4.2 Constructing a Random A-type

To introduce our construction of a random A-type we consider the size of the hypothesis space. We do this by performing the following three tasks: we consider the number of directed graphs possible for an A-type of given dimensions; we estimate a reasonable range for the delay of an A-type; and we devise a method for constructing a random A-type. This list gives us the topics for the next three subsections.

#### Combinatorics of A-type Directed Graphs

Consider an A-type that has $N$ nodes consisting of $m$ input nodes, $n$ internal nodes and $p$ output nodes. We can completely describe the directed graph of this A-type with an $N \times 2$ matrix if this matrix is an adjacency list as defined below.

**Definition 6.1** (Adjacency List)**.** Consider an A-type $A$ that has $N$ nodes consisting of $m$ input nodes, $n$ internal nodes and $p$ output nodes. Also, each node of $A$ is labelled using the integers $\{1, 2, \ldots, (N)\}$ such that: each node has a unique label; each input node is labelled with with an integer that is less than $m$; each internal node is labelled with an element of $\{(m+1), \ldots, (N-p)\}$; and each output node is labelled with an element of




$$\begin{pmatrix} -1 & -1 \\ -1 & -1 \\ 1 & -1 \\ 2 & -1 \\ 1 & 2 \\ 3 & 5 \\ 4 & 5 \\ 6 & 7 \end{pmatrix} \quad \begin{pmatrix} -1 & -1 \\ -1 & -1 \\ -1 & 1 \\ 2 & -1 \\ 2 & 1 \\ 5 & 3 \\ 4 & 5 \\ 6 & 7 \end{pmatrix}$$

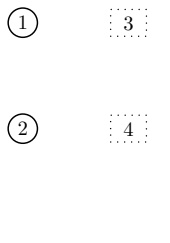
**Figure 6.4:** An A-type representing Exclusive-OR with two of the possible $2^6 = 64$ adjacency lists that prescribe its directed graph.

$\{(N-p+1), \ldots, N\}$. The *adjacency list* of $A$ for a given labelling is an $N \times 2$ matrix such that every entry in the first $m$ rows is, the placeholder, $-1$; furthermore, for every row after the first $m$ rows the two entries in the $i$th row are the labels of the nodes that are the source nodes to the node that has the label $i$. If the node with the label $i$ is a delay machine then there is only one source node, in this case the label for the source node and a place holder $-1$ are the entries for the $i$th row.

□

Figure 6.4 shows an A-type graph and two adjacency lists that prescribe that graph. The above definition may seem to give unnecessary detail but we use it to estimate the number of possible A-types for given dimensions.

Consider the set $S_A$ of all A-type graphs that have two input nodes, two internal nodes, and one output node. Note that every non-input node can be either a nand machine or a delay machine. We estimate $|S_A|$ by calculating the number of adjacency lists that represent members of $S_A$. Let $L_A$ denote an adjacency list that represents a member $A$ of $S_A$. The first two rows of $L_A$ are $\{-1, -1\}$ and $\{-1, -1\}$ because $A$ has two input nodes. Recall that an internal node is the target of a source node from $\{\text{input nodes}\} \cup \{\text{internal nodes}\}$. So, if $A$'s third node is a nand machine then the third row of $L_A$ is $\{a, b\}$ where $a, b \in \{1, 2, 3, 4\}$, and if $A$'s third node is a delay machine then the third row of $L_A$ is $\{a, -1\}$ where $a, b \in \{1, 2, 3, 4\}$. The fourth row of $L_A$ has the same form as this third row. Recall that an output node is the target of a source node from $\{\text{internal nodes}\}$. So, if $A$'s fifth node is a nand machine then the fifth row of $L_A$ is $\{c, d\}$ where $c, d \in \{3, 4\}$; also, if $A$'s fifth node is a delay machine then the fifth row of $L_A$ is $\{c, -1\}$ where $c \in \{3, 4\}$. We illustrate $L_A$ in Figure 6.5. Now we calculate the number of adjacency lists that are valid for $L_A$.




$$\begin{pmatrix} -1 & -1 \\ -1 & -1 \\ \dots & \dots \\ \dots & \dots \\ \dots & \dots \end{pmatrix} \quad \begin{array}{l} \leftarrow \{a,b\} \text{ or } \{a,-1\} \text{ where } a,b \in \{1,2,3,4\} \\ \\ \leftarrow \{c,d\} \text{ or } \{c,-1\} \text{ where } c,d \in \{1,2\} \end{array}$$

**Figure 6.5:** An illustration of how an adjacency list can be used to estimate how many A-types can be constructed that have two input nodes, two internal nodes, and one output node.

$$\underbrace{(1)}_{row\ 1} \times \underbrace{(1)}_{row\ 2} \times \underbrace{(\frac{4^2}{2}+\frac{4}{2})}_{row\ 3} \times \underbrace{(\frac{4^2}{2}+\frac{4}{2})}_{row\ 4} \times \underbrace{(\frac{2^2}{2}+\frac{2}{2})}_{row\ 5} = 300$$

We generalize this as follows. Consider the set $S$ of all A-types that have $m$ input nodes, $n$ internal nodes, and $p$ output nodes. The number of adjacency lists that are a valid representation of a member of $S$ is given by the following expression.

$$[\tfrac{(m+n)^2}{2}+\tfrac{(m+n)}{2}]^n \times [\tfrac{p^2}{2}+\tfrac{p}{2}]^p$$

Note that the above expression does not give an upper bound for the number of A-types with the given dimensions. This is because we have defined an A-type as an A-type graph with a delay $\delta$. That is, we can construct many A-types that have the same directed graph.

### Constructing a Random A-type

Here we present an algorithm for constructing a random A-type of a specified input dimension, output dimension, and number of nodes. In this project we devise and test our algorithms by constructing and running computer programs—we test our ideas '*in silico*'. We choose to encode A-types as interconnected objects rather than using adjacency lists. Therefore, our construction of a random A-type does not employ adjacency lists.

We could use adjacency lists to encode A-types. In Section 6.4.2 we introduced the adjacency list and we used this to estimate the upper bound of the number of possible directed graphs for an A-type with specified dimensions. We saw that a given adjacency list prescribes an A-type's directed graph. So a reasonable scheme for constructing a random A-type of given dimensions would be to construct an appropriately sized adjacency list.

We use interconnected objects to encode A-types rather than using adjacency lists because this makes some tasks simpler. Using interconnected objects is easier when we implement algorithms that perform specific manipulations on A-type graphs. We found that such manipulations required rather complex operations on adjacency lists (often the complexity arises from necessary relabelling of adjacency lists). The interested reader can refer to Appendix A to see a schematic of the computer programs that we constructed for this project. Our

programs represent delay machines and nand machines as objects (specifically objects in the *object oriented programming* sense), an A-type is an object that contains a set of delay machines and nand machines that reference one another. This detail can be (happily) ignored by the reader but we mention it here because it motivates the following algorithm for constructing a random A-type's graph—without using adjacency lists.

**random_Atype**$(m, n, p, d)$ constructs and returns a random A-type, $A$, of specified input and output dimensions.

Input Parameters

| *Type* | *Parameter* | *Description* |
|---|---|---|
| $\mathbb{Z}^+$ | $m$ | $A$'s input dimension. |
| $\mathbb{Z}^+$ | $n$ | Number of internal nodes of $A$. |
| $\mathbb{Z}^+$ | $p$ | $A$'s output dimension. |
| $[0, 1]$ | $d$ | Probability that a node is constructed as a delay machine. |

The algorithm has the following steps

1. *Construct A's nodes:* $m$ input nodes are constructed, $n$ internal nodes are constructed, and $p$ output nodes are constructed. Each non-input node is constructed such that the probability that it is a delay machine is $d$.
2. *Set the indegree for each internal node:* For each internal node assign the appropriate number of incoming arrows: one arrow if it is a delay machine, two arrows if it is a nand machine. The source of each of these arrows is randomly chosen from { input nodes } ∪ { internal nodes }.
3. *Set the indegree for each output node:* For each output node assign the appropriate number of incoming arrows. The source of each of these arrows is randomly chosen from { internal nodes }.
4. Return $A$.

**Table 6.8:** Constructing a random A-type.

Note that this algorithm may return an A-type that has some or all of its input nodes disconnected from the rest of the A-type. In Section 5.3.2 we devised a useful A-type with some of its input nodes disconnected from the rest of the A-type (this A-type is shown in Figure 5.9). So it is important that our algorithm is capable of generating such A-types.

### 6.4.3 Assessing an A-type: $fitness\_one$

Recall that for supervised learning we need to assess the performance of a candidate solution with respect to a set of training examples. In all three cases of *genetic_search_one* we use the same function to make such assessments. This is *genetic_search_one*'s fitness function, which we call $fitness\_one$. Although we avoid couching *blind_search_one* as an EA we still use the term fitness function to describe the means by which it assesses a candidate solution. In this section we detail $fitness\_one$.

If $fitness\_one$ accepts an A-type $A$, then the output is a number (specifically an element of $[0, 1]$) that is a function both of how well $A$ approximates the training examples and the number of nodes in $A$. If $A$ 'closely' approximates the training examples and has 'few' nodes then $A$ is deemed fit so the output of $fitness\_one$ is small[‡].

We use the normalized Hamming distance (see Section 2.3) to assess how well a candidate solution fits a set of training examples. Consider Figure 6.6, in which we show a set $T$ of three examples of Exclusive-OR, a candidate solution $A$, and a table illustrating the calculation of the normalised Hamming distance between the actual output of $A$ and the expected output for each training example. We can get a reasonable measure of the performance of $A$ with respect to $T$ by calculating the mean of the Hamming distances from all three examples; this equals $\frac{2}{4} \times \frac{1}{3} + \frac{1}{4} \times \frac{1}{3} + \frac{0}{4} \times \frac{1}{3} = \frac{1}{4}$. Note that this measure of performance could be used as a measure of fitness since a perfect match gives zero and the worst possible match gives unity.

Our fitness function also penalizes large A-types. Consider a set of Training examples $T$ and two candidate solutions $A_1$, $A_2$. If $A_1$ and $A_2$ give the identical output packets for each input packet in $T$, and if $A_2$ has 'significantly' more nodes than $A_1$, then we deem $A_1$ to be fitter than $A_2$. Via this condition we use the fitness function to 'pressure' our searches' populations into only containing small individuals. To implement this condition we devise a function $p : \mathbb{Z}^+ \times \mathbb{R}^+ \to \mathbb{R}^+$, which we call a *penalty function*. Given the size $n$ of some A-type $A$ the penalty function returns a factor that is used in the calculation of $A$'s fitness penalizing against large A-types.

Our calculation of fitness with a penalty against large A-types is as follows. Given an A-type $A$ and a set of training examples $T$ we first calculate the average $d_{av}$ of the normalised Hamming distance between $A$'s outputs and the outputs with respect to $T$. Next we scale $d_{av}$ according to $A$'s size. That is we multiply $d_{av}$ by a penalty factor $p$. To devise $p$ we consider the following three categories.

$p = 1$: $A$ has so few nodes that its performance is not penalized.

$p > 1$: $A$ has enough nodes such that its performance is penalized, the greater $A$'s size the larger the penalty.

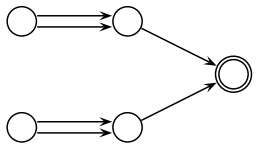
[‡]Recall that we are using a standardized (and normalized) fitness function [43, p127]. That is, low fitness values are assigned to candidate solutions that closely fit the concept being searched for.

$$T = \left\{ \left( \begin{array}{|c|c|c|c|}\hline 0&1&1&1\\\hline 0&1&1&0\\\hline\end{array} \;,\; \begin{array}{|c|c|c|c|}\hline 0&0&0&1\\\hline\end{array} \right), \right.$$

$$\left( \begin{array}{|c|c|c|c|}\hline 1&0&0&1\\\hline 0&0&1&1\\\hline\end{array} \;,\; \begin{array}{|c|c|c|c|}\hline 1&0&1&0\\\hline\end{array} \right),$$

$$\left. \left( \begin{array}{|c|c|c|c|}\hline 1&0&1&1\\\hline 0&1&0&0\\\hline\end{array} \;,\; \begin{array}{|c|c|c|c|}\hline 1&1&1&1\\\hline\end{array} \right) \right\}$$

(a) Training examples for Exclusive-OR. We denote these as $(D(i)_{in}, D(i)_{ex})$ for $i \in \{1, 2, 3\}$.



(b) Also consider the A-type $A$ whose graph is displayed above and whose delay is $\delta = 2$.

| $i$ | $D(i)_{ex}$ | $D(i)_{out}$ | $\hat{H}(D(i)_{ex}, D(i)_{out})$ |
|---|---|---|---|
| 1 | 0 0 0 1 | 0 1 1 1 | $\frac{2}{4}$ |
| 2 | 1 0 1 0 | 1 0 1 1 | $\frac{1}{4}$ |
| 3 | 1 1 1 1 | 1 1 1 1 | $\frac{0}{4}$ |

(c) Assessing $A$'s performance. For each training example we determine the normalized Hamming distance $\hat{H}(D(i)_{ex}, D(i)_{out})$ between the expected output $D(i)_{ex}$ and A's actual output $D(i)_{out}$.

**Figure 6.6:** A set of training examples, a candidate solution, and a table for that candidate solution's performance.

$p$ is constant: Because the maximum value for $A$'s fitness is unity $p$ must eventually become a constant function of $A$'s size.

For a penalty function we devise the following piecewise linear function.

$$p(n,d) = \begin{cases} 1 & \text{if } n \in \{0,1,\ldots(n_0-1)\} \\ m(n-n_0)+1 & \text{if } n \geq n_0 \text{ and } d.p < 1 \\ \frac{1}{d} & \text{if } d.p = 1 \end{cases}$$

We call $m$ the *pressure gradient* and we call $n_0$ the *penalty bound*. For example, we detail the penalty function that we employ when we search for Exclusive-OR. Because we know that there is an exact solution to Exclusive-OR that has eight nodes (as shown in Figure 6.4) we ensure that $n_0 \geq 8$. After a few investigative simulations, but in an otherwise ad-hoc fashion, we set $n_0 = 20$ and $m = \frac{1}{100}$. Consider an A-type $A$ whose output and the expected output, with respect to the search's training data, have an average Hamming distance of $d_{av}$. Our arguments of $n_0 = 20$ and $m = \frac{1}{100}$ give a penalty function $p$ with the following three properties:

1. If $A$ has fewer than 21 nodes then $p = 1$.
2. If the number of nodes in $A$ increases above 20 then $p$ increases with a gradient of $\frac{1}{100}$ until the following condition is satisfied.
3. The maximum value of $p$ is $\frac{1}{d_{av}}$.

Figure 6.7 shows the penalty function $p$ for A-types that give $d_{av} = 0.50$ (with respect to some training set). This figure also shows how the A-type's fitness depends on its size as a consequence of $p$.

In $fitness\_one$ we calculate the average $d_{av}$ of the normalised Hamming distance between $A$'s outputs and the outputs with respect to $T$, also we calculate the penalty function; the product of these two numbers gives our estimate of $A$'s fitness with respect to $T$. $fitness\_one$ is detailed in Table 6.9.

In the above example we chose the arguments $n_0 = 20$ and $m = \frac{1}{100}$ to achieve a desired penalty function. We chose these parameters knowing a small exact solution. In all four test concepts that we use in the next chapter we are privy to small exact solutions, so we can assign suitable arguments to $n_0$ and $m$. In general this is not the case. It is possible that a given choice for $n_0$ and $m$ will result in the smallest exact solutions having fitnesses greater than zero. This is acceptable because in general the termination of an EA is subjective: in general we can never know if a solution is the smallest possible solution. Eiben and Smith [39, p29] state that EAs display *anytime behaviour*. That is, an EA's search can be stopped anytime and the algorithm will have a 'solution' even if it is suboptimal.

In later chapters we revise our fitness function. A fitness function is a convenient way of imposing background information on our searches. In Chapter 8 this is one of the methods that we employ to impose symmetry constraints on our searches.

**fitness_one**$(A, T, m, n_0)$ is a fitness function that returns a measure of both how well an A-type $A$ approximates a given set of training examples $T$ and the number of nodes, $N$, in $A$.

| Input Parameters | | |
|---|---|---|
| *Type* | *Parameter* | *Description* |
| A-type | $A$ | A-type whose fitness is evaluated. |
| {training examples} | $T$ | Set of training examples used to evaluate $A$'s fitness. |
| $\mathbb{R}^+$ | $m$ | Pressure gradient. |
| $\mathbb{Z}^+$ | $n_0$ | Penalty bound. |

The algorithm has the following steps

1. *Initialize variable:*
   Set $d_{av} \leftarrow 0$, where $d_{av}$ will be the variable that holds the normalised Hamming distance between the output of $A$ and $T$.
2. *Iterate through $T$:* ForEach( training example $(D(i)_{in}, D(i)_{ex})$ of $T$ ) do
   a) *Calculate the Hamming distance:* $d_i \leftarrow \hat{H}(D(i)_{ex}, D(i)_{out})$.
   b) *Update the average:* Set $d_{av} \leftarrow (d_{av} + d_i)$.
3. *Calculate penalty factor:*
   *If a penalty is unnecessary:*
   if ( $N > n_0$ ) then do
   $p \leftarrow 1$
   *If a penalty is necessary:*
   else do
   $p \leftarrow m \times (d_{av} + 1)$.
   *Ensuring fitness is not greater than unity:*
   if ( $d_{av} \times p > 1$ ) then do
   $p \leftarrow \frac{1}{d_{av}}$.
4. Return $(d_{av} \times p)$.

**Table 6.9:** Determining the fitness of a given A-type.

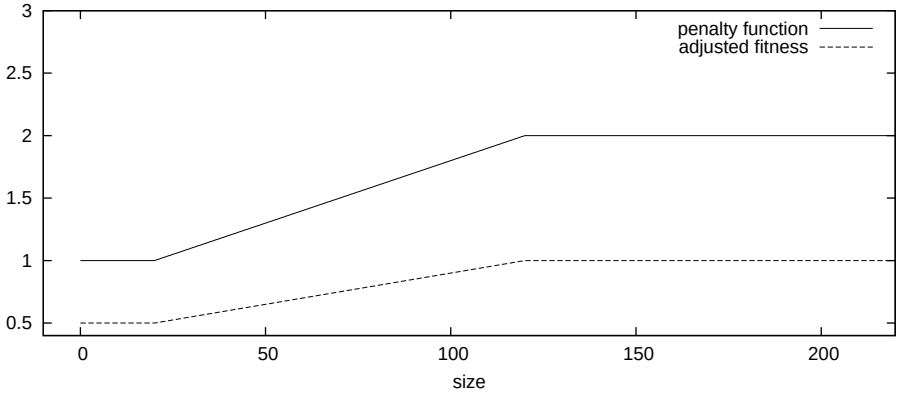



**Figure 6.7:** The solid line shows the piecewise penalty function $p$ (as prescribed in $fitness\_one$) for A-types with $d_{av} = 0.50$. The dashed line gives the resulting fitness (the product of $d_{av}$ and the penalty function). The penalty function is unity for A-types of size 20 nodes or less. For A-types with more than 20 nodes the penalty has a gradient of 0.01 unless this results in a fitness greater than unity.

### 6.4.4 Our Blind Search

Now that we have specified how we construct a random A-type and we have specified our fitness function, we can detail the random search that we use to test the performance of our EAs. We detail the algorithm here (see Table 6.10) but we delay testing this algorithm until Chapter 7.

## 6.5 An Evolutionary Algorithm Without Crossover

An EA without crossover is a useful evolutionary search scheme. Although *blind_search_one* is a special case of *genetic_search_one*, as we mentioned in the beginning of Section 6.3.1, calling it an EA seemed disingenuous because it does not employ inheritance[§]. The algorithm that we detail in this section is an EA without crossover. It is reasonable to consider such a search as an EA. There are analogous examples of biological evolution, for instance Banzhaf *et al.* [43, pp 35 - 37] detail an experiment that demonstrates the evolution of RNA molecules without the exchange of genetic material between individuals. An EA without crossover provides a test of both our blind search and more sophisticated EAs.

### 6.5.1 An Outline of *mutation_search_one*

In this subsection we outline our EA without crossover, namely *mutation_search_one*.

The outline of *mutation_search_one* is given in Table 6.11. Note that this is simply a

[§]Dawkins and Wong [77, p562] articulate this idea with the example of fire. In many regards fire is life-like; for instance, a fire's sparks may lead to daughter fires. However, it does not have any mechanism for inheritance so they classify it as not being alive.

**blind_search_one**$(m, p, l, u, f_{worst}, d)$ is a supervised learning algorithm where the hypothesis space is populated with A-types. This algorithm uses $random\_aType$ to construct a candidate solution $A_c$. How well $A_c$ approximates the concept, with respect to the set of training examples, is determined by $fitness\_function\_one$.

| Parameters for the search | | |
|---|---|---|
| *Type* | *Parameter* | *Description* |
| $\mathbb{Z}^+$ | $m$ | Input dimension $A_c$. |
| $\mathbb{Z}^+$ | $p$ | Output dimension $A_c$. |
| $\mathbb{Z}^+$ | $l$ | Lower bound of the number of nodes of $A_c$. |
| $\mathbb{Z}^+$ | $u$ | Upper bound of the number of nodes of $A_c$. |
| $[0, 1]$ | $f_{worst}$ | Maximum fitness of $A_c$ before $A_c$ is considered a solution to this search. |
| $[0, 1]$ | $d$ | Argument for $random\_aType$ that prescribes the probability of a given node in $A_c$ being a delay machine. |

The algorithm has the following steps

1. *Produce successive candidate solutions:*
   while($fitness\_function\_one(A_c) > f_{worst}$ ) do
   a) *Construct $A_c$:* Randomly choose an integer $n$ from the interval $[l, u]$.
   $A_c \leftarrow random\_aType(m, (n - m - p), p)$
2. Return $A_c$.

**Table 6.10:** A description of our blind search with A-types.

reproduction of Table 6.2 with step 2(b) omitted. This is because *mutation_search_one* is a special case of *genetic_search_one*.

If we compare our outline of *mutation_search_one*, given in Table 6.11, to our outline of *blind_search_one*, given in Table 6.7, then we can note the following three properties of *mutation_search_one* that make it different from the blind search. First, *mutation_search_one* maintains a multiset of A-types, namely the population. We use *initial_pop* to initialise this multiset¶. Second, *mutation_search_one* invokes a mutation operator. Every addition to the population requires the mutation of an existing member of the population. This is in contrast to the blind search where all hereditary information is lost between generations. Third, *mutation_search_one* invokes selection rules. In particular, we select A-types to produce mutant A-types and we select A-types to be terminated. Each selection rule can be chosen to be a probabilistic rule weighted by fitness. As we mentioned in Section 6.3.3, the analogy of biological evolution suggests that the mutant selection rule should be random and the termination rule should be fitness-based but we don't investigate this suspicion until the next chapter.

To implement *mutation_search_one* we must specify a mutation function (the fitness function is the same as that specified for *blind_search_one* and the crossover function is arbitrary); we do this next.

### 6.5.2 Introducing Mutations: *mutate_one*

Here we detail *mutation_search_one*'s mutation operator. The conjunction of this description, detail of the general case given in Section 6.3.3, and detail of $fitness\_one$ given in Section 6.4.3 completes our description of *mutation_search_one*.

Our mutation function returns an A-type whose size can differ from its input A-type. Recall that a mutation function maps an A-type to another A-type that is a 'slight' modification of the first. If a given candidate solution is mutated to produce a new candidate solution then this mutation should have the freedom to change the A-type's size for the following two reasons. First, an A-type's fitness depends on its size. So, removing nodes from an A-type may produce a solution. Second, recall that the initial population is a set of random A-types whose sizes have been randomly chosen from some interval. So, an initial population may have no A-types that have the size of any of the solution A-types. For all of the EAs described in this chapter any point in the search space is an A-type whose number of nodes is not less than some minimum. This minimum is $(D_{in} + D_{out} + 1)$, where $D_{in}$ is the input dimension of all training examples, and $D_{out}$ is the output dimension of all training examples (note that we add unity to the above sum because every A-type requires at least one internal node).

Because the only variation operator in *mutation_search_one* is mutation, if members of

¶Furthermore, in *initial_pop* each time an A-type constructs a random A-type it calls *random_Atype* which is detailed in Table 6.8

*mutation_search_one*

1. Create an initial population. That is, create a number of A-types, such that each A-type is randomly generated, and store these A-types in a multiset that we call the population.

2. Repeat until either the population contains a fit enough A-type or a maximum number of attempts have been performed. The number of these outermost iterations is recorded as the generation.

   a) Repeat a set number of times.

      i. We invoke a mutant selection rule on the population to select A-types that will be the original for a mutant.
      ii. With the original A-type we perform a mutation and we call the result the mutant A-type.
      iii. We train the mutant and put it into the population.
      iv. We invoke a termination rule on the population to select an element of the population. This element is deleted from the population.

3. Return the fittest A-type in the population.

**Table 6.11:** An outline of *mutation_search_one*. By comparing this description with that given in Table 6.2 one can quickly see that the algorithm presented here is a special case of *genetic_search_one*—the number of crossovers for each generation is zero.

the initial population are to traverse the search space then the mutation function must be able to return A-types whose size is different from the input A-type. The mutation function for *mutation_search_one*, which we call *mutate_one*, has this property.

Our description of *mutate_one* is rather piecewise, we first describe subroutines and then we present the algorithm which calls these subroutines. In particular, we first consider three mutation functions. These are mutation functions that give one of the following three effects: the size of an A-type is decreased by one; the size of an A-type is kept constant; and the size of an A-type is increased by one. Finally, we present a mutation function that incorporates all of these three mutations.

Note that when we developed our mutation functions it was important to be mindful of the rules for a valid A-type. In particular, arrows from input nodes must always enter internal nodes and arrows entering output nodes must always exit internal nodes. These rules were given in Section 4.5; however, we mentioned them here as a reminder to the reader.

### Decreasing Size

The first subroutine that we consider is a mutation operator that maps an A-type $A_{in}$ to another A-type $A_{out}$, where $A_{out}$ has one fewer nodes than $A_{in}$, and the graph of $A_{out}$ is a slight modification of the graph of $A_{in}$. We call this function *mutate_decrease*. We detail this function in Table 6.12 and we give examples of it in Figure 6.8.

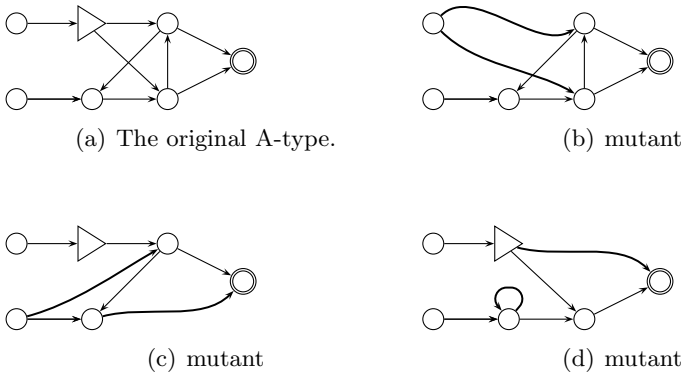
**mutation_decrease**($A_{in}$) is a mutation function that, if possible, returns an A-type that has one fewer internal nodes than the input A-type.

| Parameters | | |
|---|---|---|
| *Type* | *Parameter* | *Description* |
| A-type | $A_{in}$ | Input A-type that undergoes mutation. Recall that an A-type must have at least one internal node for it to be valid, so $A_{in}$ must have at least two internal nodes. |

The algorithm has the following steps

1. *Copy the input:* $A_{out} \leftarrow A_{in}$.
2. *Choose an internal node:* `delete_node` $\leftarrow$ a randomly chosen element of $A_{out}$'s internal nodes.
3. *Reassign the sources of the arrows exiting* `delete_node` *:*
   ForEach(arrow exiting `delete_node`) do
   *Choose a new source:* If the arrow enters an internal node then we choose the new source is assigned to an element that is randomly chosen from
   {input nodes of $A_{out}$}$\bigcup${{ internal nodes of $A_{out}$}$-$ `delete_node` }
   otherwise, the arrow enters an output node in which case the new source is assigned to an element that is randomly chosen from
   {{ internal nodes of $A_{out}$ }$-$ `delete_node` }
4. *Delete:* Delete all arrows that enter `delete_node`, then delete `delete_node`.
5. Return $A_{out}$.

**Table 6.12:** A description of a mutation function that, if possible, returns an A-type that has one fewer internal nodes than the input A-type.



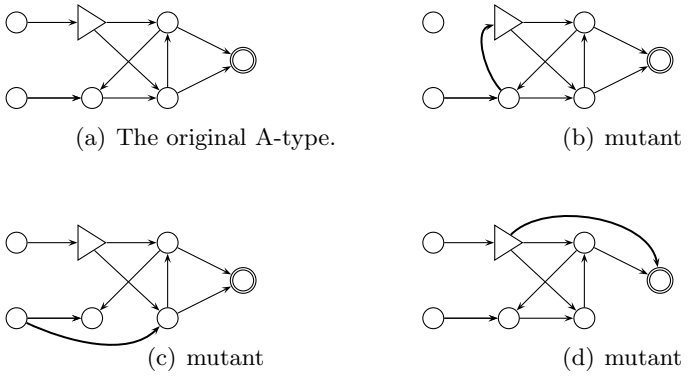
(a) The original A-type. (b) mutant

(c) mutant (d) mutant

**Figure 6.8:** Three examples of *mutate_decrease*.

### Constant Size

The second subroutine that we consider is a mutation operator that maps an A-type $A_{in}$ to another A-type $A_{out}$, where $A_{out}$ and $A_{in}$ have the same size, and the graph of $A_{out}$ is a slight modification of the graph of $A_{in}$. We call this function *mutate_constant*. We detail this function in Table 6.13 and we give examples of it in Figure 6.9.



(a) The original A-type. (b) mutant

(c) mutant (d) mutant

**Figure 6.9:** Three examples of *mutate_constant*.

### Increasing Size

The third subroutine that we consider is a mutation operator that maps an A-type $A_{in}$ to another A-type $A_{out}$, where $A_{out}$ has one more node than $A_{in}$, and the graph of $A_{out}$ is a slight modification of the graph of $A_{in}$. We ensure that for this function the added node is the source of an arrow that enters one of the original nodes of the A-type. This condition is necessary (but not sufficient) for the added node to affect the output of the A-type. We call this function *mutate_increase*. We detail this function in Table 6.14 and we give examples

**mutation_constant**($A_{in}$) is a mutation function that returns an A-type that is the same as $A_{in}$ except that the source of one of its arrows has been altered.

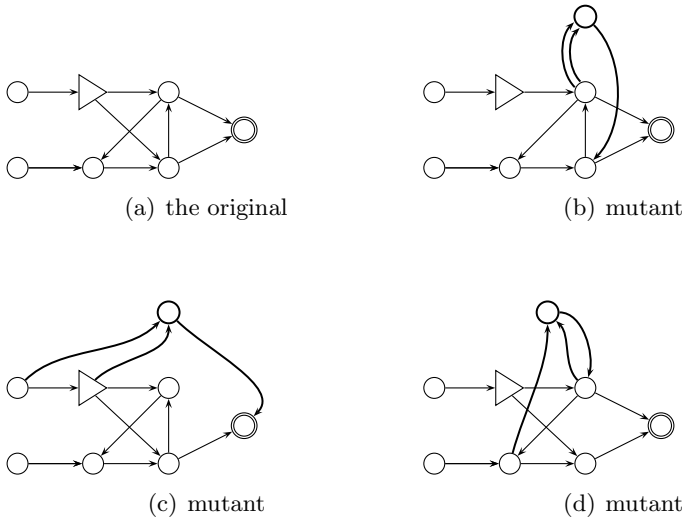
| Parameters | | |
|---|---|---|
| *Type* | *Parameter* | *Description* |
| A-type | $A_{in}$ | Input A-type that undergoes mutation. |

The algorithm has the following steps

1. *Copy the input:* $A_{out} \leftarrow A_{in}$.
2. *Choose an arrow in* $A_{out}$ *:*
   a) *Choose a target:* `target` ← a randomly chosen element of the set { input nodes of $A_{out}$ }$\bigcup${ output nodes of $A_{out}$}.
   b) *Choose a source:* `old_source` ← a randomly chosen element of the set of nodes that are sources of arrows entering `target`.
3. *Choosing a new source:* If `target` is an output node of $A_{out}$ then `new_source` ← a random element of the set of internal nodes of $A_{out}$. Otherwise, `target` is an internal node of $A_{out}$ in which case `new_source` ← a random element of the set of internal nodes of {$A_{out}$'s input nodes }$\bigcup${$A_{out}$'s internal nodes }.
4. *Replace arrow:* Delete the arrow (`old_source`,`target`) and enter the arrow (`new_source`,`target`).
5. Return $A_{out}$.

**Table 6.13:** A mutation function that returns an A-type with the same number of nodes as the input A-type. The returned A-type's graph is only a slight modification of the input A-type's graph (there is a chance that this modification is the identity).

of it in Figure 6.10.



(a) the original (b) mutant

(c) mutant (d) mutant

**Figure 6.10:** Three examples of *mutate_increase*.

### Using the Three Subroutines

We combine the above mutation functions to give a mutation function *mutate_one* that allows the size of the resulting A-type to either decrease, increase, or remain the same. In essence, this new function simply chooses, with equal probability, one of the three mutation functions defined above. However, there are two difficulties that must be catered for, these are described below.

First, if the input A-type $A_{in}$ has only one internal node then it is not a valid input for *mutation_decrease*. Consequently we introduce the condition that if $A_{in}$ has only one internal node then we choose, with equal probability, either *mutation_constant* or *mutation_increase*.

Second, our mutation function may lead to very large A-types. Not only will large A-types be penalised by the fitness function but also, in practice, we found that when our searches maintained a population that contained large A-types the searches took a long time to complete. We cater for this by introducing an upper bound into our mutation function. In our searches for Exclusive-OR it is easy to set an upper bound that does not exclude every solution because we have a solution that has eight nodes—as shown in Figure 6.4

### Selecting Size

When devising *mutate_one* we decided to restrict the output of *mutation_increase* so that it returns A-types with two restrictions: the out-degree of the added node is one, and the probability that the added node is a delay machine is 0.20. That is when *mutate_one* calls the function to increase the size of an A-type $A_{in}$ it calls $mutation\_increase(A_{in}, 0.20, 1)$.

**mutation_increase**$(A_{in}, d, n)$ is a mutation function that returns an A-type that has one more internal node than the input A-type. The returned A-type has at least one edge from this new node.

| Parameters for the Initial Population | | |
|---|---|---|
| *Type* | *Parameter* | *Description* |
| A-type | $A_{in}$ | Input A-type that undergoes mutation. |
| $[0, 1]$ | $d$ | Probability that a node is constructed as a delay machine. |
| $\mathbb{Z}^+$ | $n$ | Outdegree of the added node. |

The algorithm has the following steps

1. *Copy the input:* $A_{out} \leftarrow A_{in}$.
2. *Make new node:* Construct a new node `insert_node` such that the probability that it is a delay machine is $d$.
3. *Choose source(s) for the new node:* Source nodes for `insert_node` are elements randomly chosen from the set { input nodes of $A_{out}$}$\bigcup${ internal nodes of $A_{out}$}. If `insert_node` is a delay machine then only one such element is required; otherwise, `insert_node` is a nand machine and two such elements are required.
4. *Insert $n$ arrows:*
   Repeat $n$ times
   a) *Choose a target node:* `target` $\leftarrow$ a randomly selected element of { internal nodes of $A_{out}$}$\bigcup${ output nodes of $A_{out}$}.
   b) *Make a vacancy for target node:* Randomly select an arrow that enters `target` and delete that arrow from $A_{out}$.
   c) *Insert:* Insert the arrow (`insert_node` , `target`).
5. Return $A_{out}$.

**Table 6.14:** A mutation function that returns an A-type that has one more internal node than the input internal node.

### 6.5.3 The Algorithm

We can now detail our mutation function, we do this in Table 6.15. This completes our presentation of *mutation_search_one*.

**mutate_one**$(A_{in}, u)$ is a mutation operator that returns an A-type whose graph is a modification of the input A-type. The return A-type may have one more or one fewer nodes than the input A-type.

| Parameters | | |
|---|---|---|
| *Type* | *Parameter* | *Description* |
| A-type | $A_{in}$ | Input A-type. |
| $\mathbb{Z}^+$ | $u$ | If $A_{in}$ has at least $u$ (upper bound) internal nodes then the mutation called does not increase the size of the A-type. |

The algorithm has the following steps

1. *Choosing the type of mutation:*

   $\mathtt{N} \leftarrow$ the number of internal nodes in $A_{in}$.

   if(N = 1) then choose, with equal probability, either

   $A_{out} \leftarrow mutation\_constant(A_{in})$

   or

   $A_{out} \leftarrow mutation\_increase(A_{in}, 0.20, 1)$

   otherwise do

   if($\mathtt{N} \geq u$) then choose, with equal probability, either

   $A_{out} \leftarrow mutation\_decrease(A_{in})$

   or

   $A_{out} \leftarrow mutation\_constant(A_{in})$

   otherwise choose, with equal probability, either

   $A_{out} \leftarrow mutation\_decrease(A_{in})$

   or

   $A_{out} \leftarrow mutation\_constant(A_{in})$

   or

   $A_{out} \leftarrow mutation\_increase(A_{in}, 0.20, 1)$.

2. Return $A_{out}$.

**Table 6.15:** A description of the mutation function that we use for *mutation_search_one* and for the general case *genetic_search_one*.

## 6.6 An Evolutionary Algorithm With Crossover

Here we detail our crossover operator for A-types. This operator employs graph theoretic concepts in an attempt to devise a useful recombination operator. This section completes our description of *genetic_search_one*.

### 6.6.1 Our Approach

In most instances of biological breeding the resulting child is very similar to its parents; that is, the child's fitness is close to that of its parents. This is achieved by speciation and homologous crossover. With regard to 'breeding' A-types we hypothesize that there exists a crossover scheme that usually returns a child that is similar to its parents. That is, we hypothesize that if two parent A-types have similar fitnesses then there exists a method of performing the following three tasks: choosing a subgraph from each of the parent A-types; exchanging the subgraphs; and reconnecting nodes giving children that are valid A-types, such that the fitnesses of the children are similar to that of their parents. In an attempt to verify this claim we implement a somewhat intricate crossover function.

Our implementation of crossover entails the exchange of small 'chunks' of two A-types' graphs. In our EA we want crossover to be the fine adjustment and mutation to be the coarse jumps through the hypothesis space. This is seen in biology: a child is usually similar to its parents, but a mutation can lead to a large (usually disadvantageous) variation. For our crossover operator we carefully specify the 'chuncks' that are exchanged by employing the notion of a subgraph (defined in Section 2.2). Also, we are particular about how a 'chunk' is connected into an A-type's graph. We require that all nodes in an exchanged subgraph are connected to one another. Also, we require that an exchanged subgraph be a radial subgraph. Finally, when a subgraph is inserted into the complement of another subgraph the necessary rewiring of the child subgraph is restricted to the boundaries of the subgraphs.

### 6.6.2 Introducing Crossover: *crossover_one*

Here we provide an outline of our crossover operator, which we call *crossover_one*. This is a function that maps a pair of 'parent' A-types to a 'child' A-type. It takes a radial subgraph from one parent and the complement of a radial subgraph from the other parent, and connects the boundaries of these subgraphs. We suspect that when we exchange subgraphs it is useful to preserve as much of the parents' topology as possible. Specifically we make two hypotheses: a radial subgraph is preferable to a random subgraph, and reconnection between boundaries is preferable to random reconnection.

Now we provide motivation for the first hypothesis. Consider an A-type graph $G$ and the following two subgraphs of $G$: a radial set $R_G$ with $n$ nodes that has been randomly selected from the set of radial subgraphs of $G$'s internal nodes that has $n$ nodes; and a subgraph $S_G$

*crossover_one*

Given two A-types, one which we call the mother ♀ and the other the father ♂.

1. *Choose an acceptor:* Choose a radial set that is a subset of ♀'s internal nodes. We call this radial set the acceptor, $A$.
2. *Determine boundary:* Determine $A$'s distal boundary $B_A$.
3. *Choose a donor:* Choose a radial set that is a subset of ♂'s internal nodes. We call this radial set the donor, $D$.
4. *Determine boundary:* Determine $D$'s distal boundary $B_D$.
5. *Construct child:* Construct the union of the compliment of $A$ with $D$, we call this the child.
6. *Start connecting:* Arrows are inserted into vacancies in $B_D$ the source of each arrow is randomly chosen from $B_A$.
7. *Finish connecting:* Arrows are inserted into vacancies in $B_A$ the source of each arrow is randomly chosen from $B_D$.
8. Return the child.

**Table 6.16:** An outline of our crossover operator.

that also has $n$ nodes but $S_G$ is constructed by randomly selecting nodes from $G$'s internal nodes. In most cases $R_G$ will have more arrows than $S_G$ so $R_G$ preserves more of $G$'s topology than $S_G$ does.

Now we provide motivation for the second hypothesis. When we perform crossover we remove an acceptor subgraph from the mother and replace it with a donor subgraph from the father. When we reconnect subgraphs, if we choose source arrows exclusively from boundary nodes rather than from any child node then the donor has a greater similarity to its parent's graph; also, the complement of the acceptor has a greater similarity to its parent's graph.

We outline *crossover_one* in Table 6.16 and we give a simple cartoon of the algorithm in Figure 6.11.

### 6.6.3 Describing our Crossover

We present a description of *crossover_one* in Table 6.17. To clarify this presentation we employ two subroutines which are detailed in Table 6.18 and Table 6.19.

**crossover_one**( ♀, ♂, $p_{max}$ ) is a crossover algorithm that accepts two parent A-types and returns a child A-type. An outline of this algorithm is given in Table 6.16. Note that in this description the procedure of choosing subgraphs is relegated to the subroutine *choose_subgraph* and the construction of the child to *construct_child*.

| Parameters | | |
|---|---|---|
| *Type* | *Parameter* | *Description* |
| A-type | ♀ | Mother. |
| A-type | ♂ | Father. |
| $[0, 100]$ | $p_{max}$ | Maximum size of a subgraph as a percentage of the size of a parent's internal nodes |

The algorithm has the following steps

1. *Determine the acceptor:* We choose an acceptor subgraph from the mother: `A` ← *make_subgraph*( ♀, $p_{max}$).
2. *Determine the acceptor's boundary:* We determine the distal boundary of $A$ and remove any nodes that are input nodes of ♀. We call the resulting set of nodes $B_A$. If $B_A$ is empty then we repeat the above procedure until $B_A$ contains at least one node.
3. *Determine the donor:* We choose a donor subgraph from the father: `D` ← *make_subgraph*( ♀, $p_{max}$).
4. *Determine the donor's boundary:* We determine the proximal boundary of $D$. We call the resulting set of nodes $B_D$. If $B_D$ is empty then we repeat the above procedure until $B_D$ contains at least one node.
5. *Construct the child by copying* ♀ *and replacing A with D:* `child` ← *construct_child*((♀ $-A$), $B_A$, $D$, $B_D$ ).
6. Return `child`.

**Table 6.17:** A description of *crossover_one*. Note that this description employs subroutines that are detailed in the next two tables.

**choose_subgraph**( $P$, $p_{max}$ ) is an algorithm that returns a radial set that is a subgraph of the parent A-type $P$. The size of the subgraph is either unity or no greater than $p_{max}$ percent of the size of $P$'s internal nodes.

| Parameters | | |
|---|---|---|
| *Type* | *Parameter* | *Description* |
| A-type | $P$ | Parent. |
| $[0, 100]$ | $p_{max}$ | Upper bound of the size of the returned radial set as a percentage of the size of $P$'s internal nodes. |

The algorithm has the following steps

1. *Choose the centre:* From $P$ we randomly choose an internal node $c_{rad}$—this will become the a centre of a radial set.
2. *Choose the size:* We randomly choose the size of the radial set, $s_{rad}$ from the set $\{1, \ldots, n\}$, where $n$ is the value of $p_{max}$ $\times$ (number of internal nodes in $P$ ) rounded down to an integer or set to unity if zero is the result of the rounding.
3. *Choose the radial set:* We randomly choose an element, which we call `subgraph`, from the set of radial sets with centre $c_{rad}$ and of size $s_{rad}$.
4. Return `subgraph`.

**Table 6.18:** A description of a subroutine for *crossover_one*. It details how we choose a radial set from an A-type's internal nodes.

**construct_child**( $C_A$, $B_A$, $D$, $B_D$ ) is an algorithm that, given two subgraphs of A-types and given boundaries within those subgraphs, returns an A-type constructed from those A-types. The returned A-type is constructed by inserting arrows between members of the boundaries.

| Parameters | | |
|---|---|---|
| *Type* | *Parameter* | *Description* |
| subgraph | $C_A$ | Complement of the acceptor subgraph of ♀. |
| set of Nodes | $B_A$ | Distal boundary of the acceptor subgraph. |
| subgraph | $D$ | Donor subgraph of ♂. |
| set of Nodes | $B_D$ | Proximal boundary of the donor subgraph. |

The algorithm has the following steps

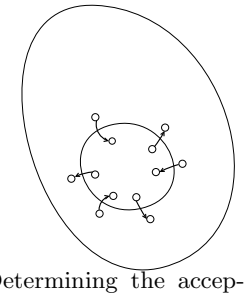
1. *Create a new subgraph:* `child` $\leftarrow C_A + D$.
2. *Set arrows into D:* We check the indegree of each node $n$ in $B_D$. If $n$ does not have the correct indegree (where the correct indegree for nand machines is 2 and the correct indegree for delay machines is 1) then a node $m$ is randomly selected from $B_A$ and an arrow is inserted from $m$ to $n$ until the indegree of $n$ is correct.
3. *Set arrows into $C_A$:* This step is similar to the previous step. We check the indegree of each node $n$ in $B_A$. If $n$ does not have the correct indegree then a node $m$ is randomly selected from $B_D$ and an arrow is inserted from $m$ to $n$ until the indegree of $n$ is correct.
4. Return `child`.

**Table 6.19:** A subroutine for *crossover_one*. This details how we interconnect the donor and the compliment of the acceptor.

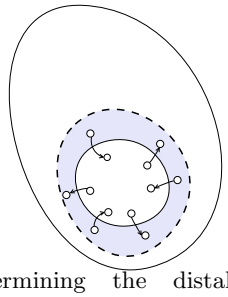
We present a concrete example of *crossover_one* in Figure 6.12. There we illustrate two parent A-types ♀, ♂ and two radial subgraphs $A$, $D$ in those parents. Nodes in the distal boundary of the acceptor are displayed with hatching ( ⊘ ). Nodes in the proximal boundary of the donor are displayed with cross-hatching ( ⊗ ). Notice that ♀, ♂, and the resulting child all have the same input dimensions, and they all have the same output dimensions, yet the child's size differs from the size of either parent.

## 6.7 Conclusion

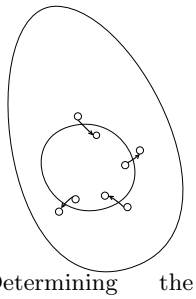
In this chapter we detail an evolutionary algorithm (EA) that evolves a population of A-types. Each individual in the population is an A-type graph and a range of delays; that is, each individual is a set of A-types all of which have the same graph. We detail how we construct a random A-type for the initial population. We detail the EA's mutation operator which accepts an A-type and produces another, mutant, A-type whose size may differ by one. We detail the EA's crossover operator, where we use graph theoretic considerations to try to construct a useful crossover operator. This chapter is rather detailed and this reflects the effort required to encode the EA (in particular the EA's crossover operator) into a computer program. Two special cases of our EA, also presented in this chapter, enable us to investigate the EA's utility. In the next chapter we experimentally conduct such an investigation.



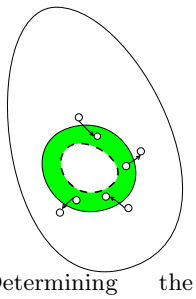
(a) Determining the acceptor $A$, which is a radial subgraph of the mother's internal nodes.



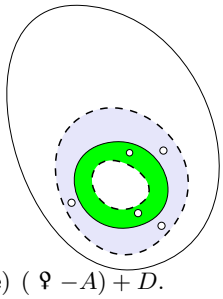
(b) Determining the distal boundary of $A$.



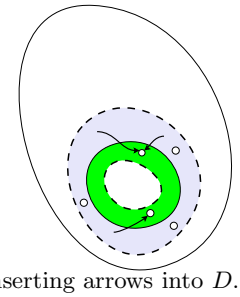
(c) Determining the donor $D$, which is a radial subgraph of the father's internal nodes.



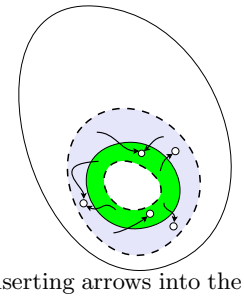
(d) Determining the proximal boundary of $D$.



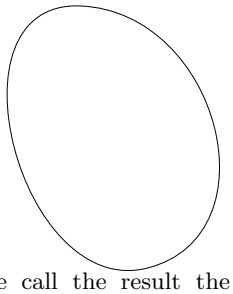
(e) $(\,\text{♀} - A) + D$.



(f) Inserting arrows into $D$.

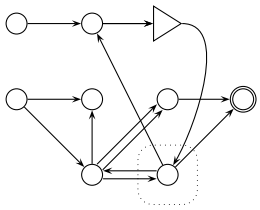


(g) Inserting arrows into the complement of $A$.

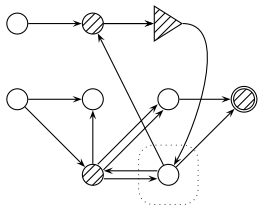


(h) We call the result the child.

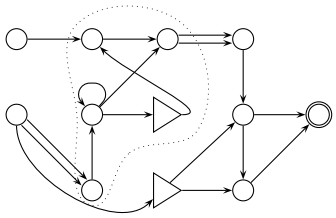
**Figure 6.11:** A schematic of *crossover_one*.



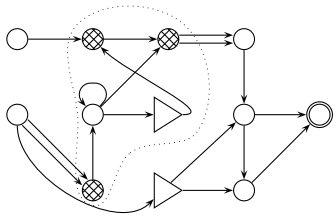
(a) The mother ♀ and its acceptor subgraph $A$.



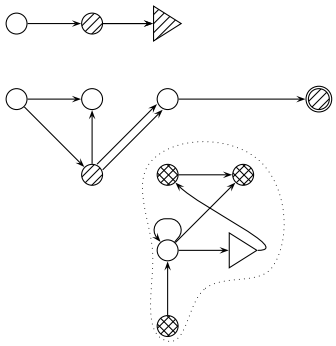
(b) Determining the boundary of $A$.



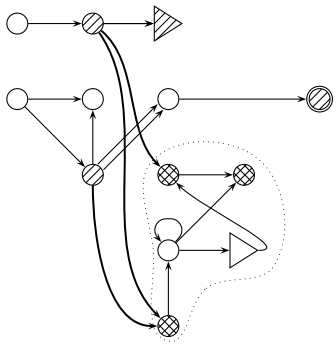
(c) The father ♂ and its donor subgraph $D$.



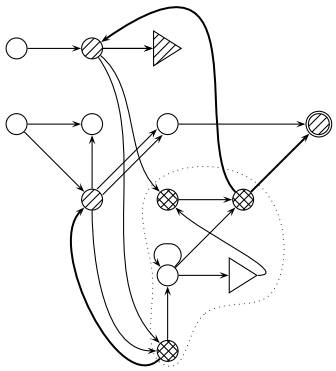
(d) Determining the boundary in $D$.



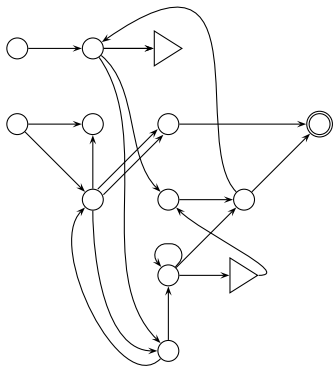
(e) $(\ ♀ - A) + D$.



(f) Inserting arrows into $D$.



(g) Inserting arrows into the boundary of $A$.



(h) The child.

**Figure 6.12:** A concrete example of *crossover_one*.

# 7 Simulations with A-types

## 7.1 Aims of this Chapter

In this chapter we present the results of computer simulations of our EAs with populations of A-types. We devised, implemented, and tested a computer program (which is outlined in Appendix A) that uses the algorithms presented in Chapter 6 to evolve a population of A-types using training data for a particular concept. In this chapter we detail the results of using this computer program for two investigations: we investigate whether delay machines are necessary, and we compare the performance of *blind_search_one*, *mutation_search_one*, and *genetic_search_one* when they are employed to search for one of four simple concepts.

## 7.2 The necessity of Delay Machines

In this section we describe the computer simulations that we conducted to investigate whether our sequential A-types require delay machines. That is, we describe the experiments that we conducted to test Claim 5.3 and Claim 5.4.

### 7.2.1 Experimental Method

For each claim we searched for A-types that represented particular concepts; the failure of these searches supports our claims. We verify Claim 5.3 if we prove that there are no A-types without delay machines that represent columnwise Exclusive-OR. So we searched for such an A-type and the failure of this search supports Claim 5.3. We verify Claim 5.4 if we prove that there are no A-types without delay machines that have an odd delay and that represent columnwise identity. Again, we searched for such an A-type and the failure of this search supports Claim 5.4. In both cases we require 'suitable' training data. We discuss this next.

Consider searching for an A-type that represents some concept $c$. We assume that if a training set is suitable for a search for $c$ that only uses A-types with delay machines then it is also suitable for a search for $c$ that only uses A-types without delay machines. We articulate this in the following claim.

**Claim 7.1.** Consider a search $\mathcal{S}$ that employs a training set $T$ to search for a concept $c$. Also, consider the following two hypothesis spaces: space $H_1$ that only contains A-types with delay machines, and with a ratio $r_1$ of solutions to non-solutions; and space $H_2$ that only

$$\left\{ \left( \begin{array}{|c|c|c|}\hline 1&0&1\\\hline 0&0&0\\\hline\end{array} \;,\; \begin{array}{|c|c|c|}\hline 1&0&1\\\hline\end{array} \right),\; \left( \begin{array}{|c|c|c|}\hline 0&0&1\\\hline 0&1&0\\\hline\end{array} \;,\; \begin{array}{|c|c|c|}\hline 0&1&1\\\hline\end{array} \right) \right\}$$

(a) Two examples of columnwise Exclusive-OR.

$$\left\{ \left( \begin{array}{|c|c|c|c|c|}\hline 0&0&1&0&1\\\hline 0&1&0&0&0\\\hline\end{array} \;,\; \begin{array}{|c|c|c|c|c|}\hline 0&1&1&0&1\\\hline\end{array} \right) \right\}$$

(b) A single example of columnwise Exclusive-OR.

**Figure 7.1:** Two training sets for Exclusive-OR that contain a similar amount of information.

contains A-types without delay machines, and with a ratio $r_2$ of solutions to non-solutions. If $p \times r_1$ is the probability that $\mathcal{S}$ discovers a solution within $H_1$, then $p \times r_2$ is the probability that $\mathcal{S}$ discovers a solution within $H_2$.

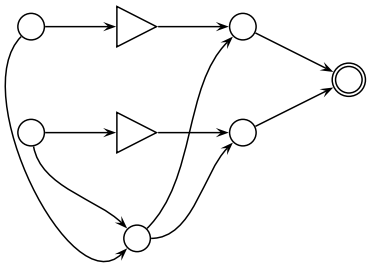
□

So when we investigate each claim from Chapter 5 our experimental method has the following three steps. First, we determine a training set $T$ that enables the discovery of solutions for over half of the searches performed using A-types with delay machines. Second, using $T$ we perform many trials using A-types with delay machines and the same number of trials with A-types without delay machines. Third, if these searches result in numerous solution A-types with delay machines and no solution A-types without delay machines then this null result supports the tested claim.

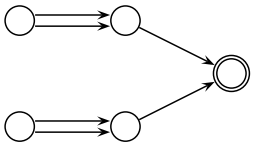
Now we outline our choice of training data for our simulations. These simulations employ training sets that contain a single example. Because we use sequential A-types, given a training set with several examples we can devise another training set with a single example that contains a similar amount of information. For example, Figure 7.1 shows two training sets for columnwise Exclusive-OR that have a similar amount of information—one can easily be constructed from the other.

If the reader examines Figure 7.1 then they can see that we contrived these sets to fail to discriminate between Inclusive-OR and Exclusive-OR. For instance, both A-types shown in Figure 7.2 exactly fit the training set. This illustrates that a training set should be a 'representative sample' of the concept being learnt. Consequently, for our simulations the input data packets of our training examples are random data packets. That is, we use random examples as defined next.

**Definition 7.1.** Consider a columnwise function $f : M[m, n] \to M[m, p]$, where $m, n, p \in \mathbb{Z}^+$. We construct a *random example* of $f$ which is a pair $(D_{in}, f(D_{in}))$ by assigning every entry



(a) An A-type with a delay $\delta = 3$.



(b) An A-type with a delay $\delta = 2$.

**Figure 7.2:** Two A-types that exactly fit the examples in Figure 7.1.

in $D_{in}$ a randomly selected element of $\mathbb{Z}_2$, and then determining $f(D_{in})$.

□

If a training example $e$ of some concept $c$ is very long then it is likely that an A-type that fits $e$ also represents $c$ for any arbitrarily long example. If this is the case then say that the A-type is an *exact* solution to a search for A-types that represent $c$. Searching with a long training example ensures that most solutions are exact but this is computationally expensive. We use smaller training examples and test the exactness of each solution. For the simulations in this section if a solution represents a random example of $c$ with an input data packet that has a length of 1000 then we deem that solution to be exact. Our desired training set is the shortest random example that leads to exact solutions for over half of the searches using A-types with delay machines.

### 7.2.2 Searching for Exclusive-OR without Delays

First, we searched for an appropriate length for our random training example. We considered a range of example lengths and for each length we conducted twenty trials. For each trial we employed *mutation_search_one*[*] to search for A-types with delay machines that fit the training data. These solutions were then tested against a random Exclusive-OR training example of length 1000. If a solution fitted that example then it was considered to be an exact solution. The results of these trials are shown in Figure 7.3. For short training examples non-exact solutions were prevalent, as the length of the examples increased exact solutions were more frequent. Inexact solutions were infrequent for training examples with data packets longer than about 50; however, sometimes the search would exceed 500,000 attempts and be terminated without solution. Because we require a training set that often gives a solution (when a solution exists), rather than the optimal training set, we simply choose to use a training set length that often results in exact solutions. From the results shown in Figure 7.3,

[*]We used an EA because we found it more efficient than a blind search. We discovered this in a rather informal fashion—in later sections we formally compare the the performance of our algorithms.

we chose to conduct this investigation using random examples of Exclusive-OR of length 50.

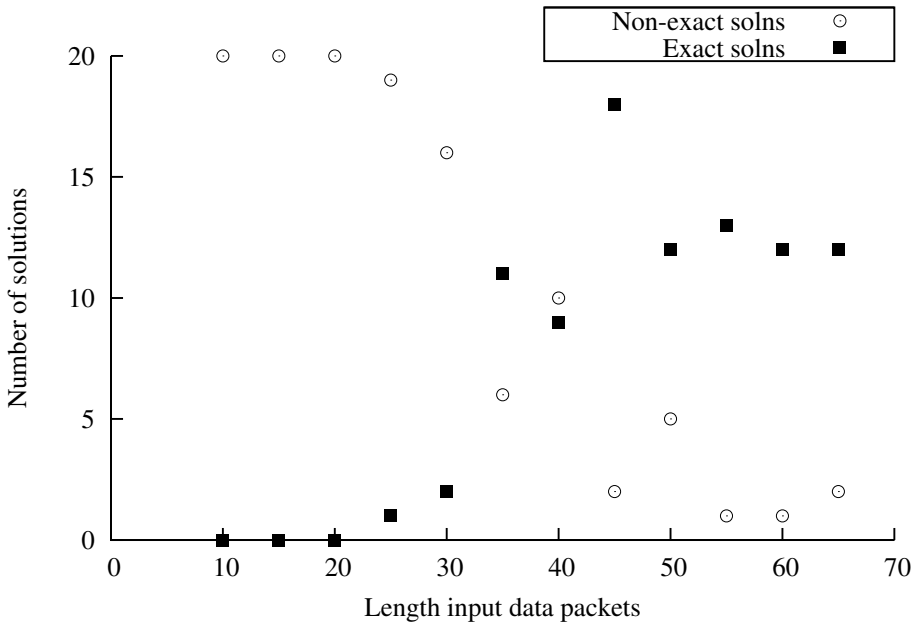



**Figure 7.3:** Determining a suitable length of a single training example for A-types that represent columnwise Exclusive-OR. For each length we performed 20 trials . Each trial either found an exact solution, found a non-exact solution or terminated without solution (after 500,000 attempts). From these results we see that training data with data packets of length 50 is likely to give an exact solution; consequently, we use such training data.

Second, we search for A-types that represent columnwise Exclusive-OR. We performed 200 trials using A-types without delay machines and 200 trials with A-types with delay machines. Any search that exceeded 500,000 generations without returning a solution was deemed a failure and was terminated. The results of these trials are shown in Table 7.1.

| | Frequency of Solutions | |
|---|---|---|
| Solution Type | Without Delay Machines | With Delay Machines |
| exact | 0 | 121 |
| non-exact | 3 | 32 |
| failure | 197 | 47 |

**Table 7.1:** Evidence that A-types with a serial input scheme require delay machines to represent Exclusive-OR.

| Solution Type | Frequency of Solutions |
|---|---|
| exact and even delay | 853841 |
| exact and odd delay | 0 |
| non-exact | 146159 |
| failure | 0 |

**Table 7.2:** Using *mutation_search_one* to search for columnwise identity. All exact solutions to this search have an even delay.

| Solution Type | Frequency of Solutions |
|---|---|
| exact and even delay | 962815 |
| exact and odd delay | 0 |
| non-exact | 37185 |
| failure | 0 |

**Table 7.3:** Using *blind_search_one* to search for columnwise identity. All exact solutions to this search have an even delay.

In conclusion, we failed to find at any A-type without delay machines that exactly represents columnwise Exclusive-OR. This null result supports Claim 5.3.

### 7.2.3 Searching for Identity with Odd Delay

First, we decided upon suitable training data. We choose our training data to be a single random example of columnwise identity. Since each trial was significantly faster than each trial of the previous Exclusive-OR searches we were less systematic in determining an appropriate length for our training examples. After trials and errors we chose to use examples of length 20. We deemed a solution to be exact if it represented a random example of columnwise identity whose length was 1000. Because the simulations had a large proportion of exact solutions, and because we were able to perform millions of trials, our choice of training data seems reasonable.

We performed $10^6$ trials using A-types without delay machines and recorded the delay of every solution. Any search that exceeded 500,000 attempts was terminated and deemed a failure. We performed our simulations with two distinct search methods: *blind_search_one* and *mutation_search_one*. Since Claim 5.4 is easy to test experimentally we took the opportunity to check whether our results are an artifact of the search method employed by conducting trials with both algorithms. Our results are presented in Table 7.2 and Table 7.3. In conclusion, we failed to find any A-type that has an odd delay and (exactly) represents columnwise identity. This null result supports Claim 5.4.

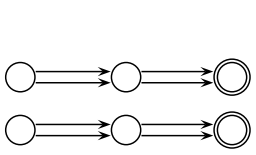
## 7.3 Description of Benchmark Concepts

In this section we introduce the four concepts that we use to compare the performance of *blind_search_one*, *mutation_search_one*, and *genetic_search_one*.

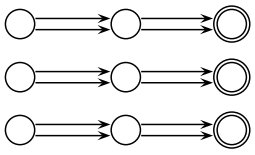
### 7.3.1 Identity

The first concept that we present is the $n$-identity function. In Section 5.3 we defined identity; now we generalize this concept by defining the $n$-identity function. This is a Boolean function $id_n : (\mathbb{Z}_2)^n \to (\mathbb{Z}_2)^n$ where $id_n(x) = x$ for all $x \in (\mathbb{Z}_2)^n$. So identity defined in Section 2.3 is 1-identity as defined above.

Two properties of $n$-identity make it a suitable benchmark concept. First, we can easily construct an A-type that represents $n$-identity. For example, the A-types illustrated in Figure 5.3, Figure 7.4(a), and Figure 7.4(b) represent 1-identity, 2-identity, and 3-identity respectively. In each of these examples the A-type represents columnwise $n$-identity; consequently, the A-type also represents $n$-identity in the clamped scheme. Second, from initial ad hoc investigations we know that our simulations quickly find A-types that represent $n$-identity for small $n$ values, such as $n \in \{1, 2, 3, 4\}$. So, we suspect that performing many trials over a range of small $n$ values is computationally inexpensive.



(a) An A-type, with a delay of $\delta = 2$, that represents columnwise 2-identity.



(b) An A-type, with a delay of $\delta = 2$, that represents columnwise 3-identity.

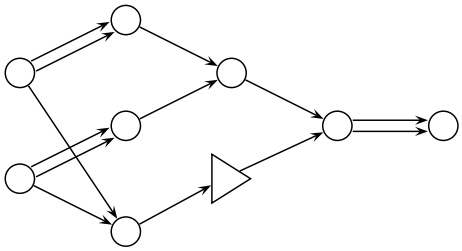
**Figure 7.4:** Two A-types that represent columnwise $n$-identity.

### 7.3.2 Parity

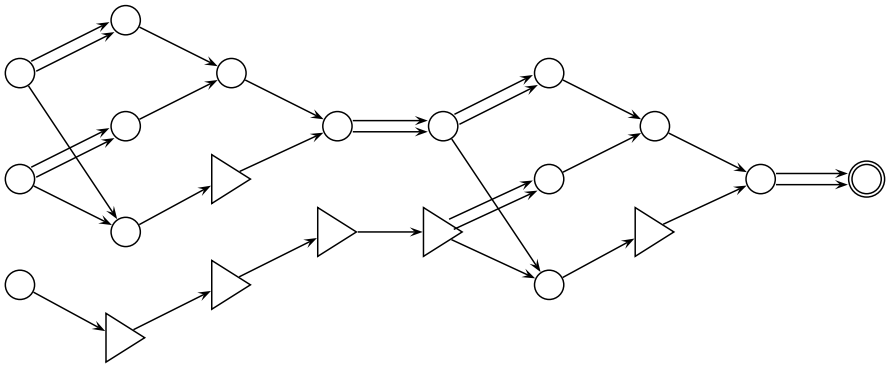
The second concept that we present is the $n$-parity function. In Section 5.3.4 we defined Exclusive-OR; now we generalize this concept by defining the $n$-parity function. This is a Boolean function $p_n : (\mathbb{Z}_2)^n \to \mathbb{Z}$ where $n \in \mathbb{Z} \mid n > 1$ such that for any $x \in (\mathbb{Z}_2)^n$

$$p_n(x) = \begin{cases} 1 & \text{if } x \text{ has an odd number of 1's} \\ 0 & \text{otherwise} \end{cases}$$

So Exclusive-OR is 2-parity.



(a) An A-type, with a delay $\delta = 4$ that represents columnwise 2-parity.



(b) An A-type, with a delay $\delta = 8$ that represents columnwise 3-parity.

**Figure 7.5:** Constructing an A-type that represents columnwise 3-parity by composing two A-types that represent 2-parity.

We can construct an A-type that represents any given columnwise $n$-parity function. In Table 7.4 we prove that Exclusive-OR is associative. For any finite $n \in \mathbb{Z} \mid n \geq 2$ we can construct $n$-parity by composing $(n-1)$ Exclusive-OR functions. That is, $p_n(x_1, x_2, \ldots, x_n) = x_1 \oplus x_2 \oplus \ldots \oplus x_n$. In Figure 7.5(a) (which we reproduced from Figure 5.11(c)) we illustrate an A-type $A$ that represents columnwise Exclusive-OR. So for any finite integer $n > 2$ we can construct an A-type that represents columnwise $n$-parity by composing $(n-1)$ copies of $A$. For example, in Figure 7.5(b) we construct an A-type that represents columnwise 3-parity.

Two properties of $n$-parity make it an suitable benchmark concept. First, as demonstrated above, for any given finite integer $n$ greater than unity we can construct an A-type that represents columnwise $n$-parity by composing $(n-1)$ copies of the A-type shown in Figure 7.5(a). Second, an A-type that represents columnwise 2-parity must contain delay machines—we investigated this assertion in Section 7.2.

### 7.3.3 Multiplexing

The third concept that we present is the $n$-multiplexer. A multiplexer is a type of switch; it allows the source of a single output line to be selected from several input lines. Brown and

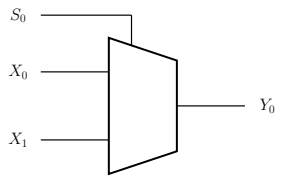
| $A$ | $B$ | $C$ | $A \oplus B$ | $(A \oplus B) \oplus C$ | $B \oplus C$ | $A \oplus (B \oplus C)$ |
|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 1 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 | 1 | 1 |
| 0 | 0 | 1 | 0 | 1 | 1 | 1 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |

**Table 7.4:** A truth table proving that Exclusive-OR is associative.

Vranesic [78, section 6.1] provide an excellent treatment of multiplexers and we borrow from them for this presentation. First we consider a 2-multiplexer. Figure 7.6 gives a schematic and the truth table for the 2-multiplexer. The 2-multiplexer has two data inputs, $x_0$ and $x_1$, a single selection input, $s_0$, and a single data output, $y_0$. From the truth table shown in Figure 7.6(b) we see that if $s_0 = 0$ then the output $y_0$ is the same as the data input $x_0$; also, if $s_0 = 1$ then the output $y_0$ is the same as the data input $x_1$. That is, changing the value of $s_0$ switches the output between $x_0$ and $x_1$. Now let us consider the 3-multiplexer whose schematic is given in Figure 7.7. The 3-multiplexer has three data inputs, $x_0$, $x_1$, and $x_2$, two selection inputs, $s_0$ and $s_1$, and a single data output, $y_0$. From the truth table shown in Figure 7.7 we see that the values of the selection inputs prescribe which data input is 'connected' to the output $y_0$. The pair $(s_0, s_1)$ selects which data input line is the same as the output, namely $(0, 0)$ selects $x_0$, $(0, 1)$ selects $x_1$, and $(1, 0)$ selects $x_2$. For any n-multiplexer (where $n \in \mathbb{Z} \mid n > 1$) the selector input pins are read as a binary representation of an integer, this integer represents a data input line. So a $n$-multiplexer requires $log_2(n)$ selector pins (rounded up to the next integer).

We can construct an A-type that represents any given $n$-multiplexer. This construction is similar to our construction of $n$-parity from $(n - 1)$ 2-parity machines. In Figure 7.8 we demonstrate how two 2-multiplexers can be composed to construct a 3-multiplexer. Furthermore, we illustrate an A-type that represents columnwise 2-multiplexer and the composite A-type that represents columnwise 3-multiplexer. In general, for an arbitrarily large finite $n > 2$ we can construct an A-type that represents columnwise $n$-multiplexer by composing A-types that represent 2-multiplexer.

Two properties of $n$-multiplexer make it an suitable benchmark concept. First, as shown above, given a finite integer $n$ greater than unity we can construct an A-type that represents $n$-multiplexer. Second, many researchers have applied EAs to the task of discovering multiplexers. This started with Wilson [79] and others have also investigated this task, for




(a) A schematic of the 2-multiplexer.

| $s_0$ | $x_0$ | $x_1$ | $y_0$ |
|---|---|---|---|
| 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 1 |
| 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 0 |

(b) A truth table for the 2-multiplexer.

**Figure 7.6:** The 2-multiplexer.

example Koza [49, ch 7], Butz [80, ch 3]. In particular, Bull and Preene [74] investigated searches for multiplexers using A-types.

### 7.3.4 Carry

The fourth concept that we present is the $n$-carry. We contrived this concept to investigate a sequential task that has no obvious clamped special case. The $n$-carry concept is a set of functions $c_n = \{(M[1,l] \rightarrow M[(l-n+1),n]\}$ where $l$ and $n$ are positive integers such that $n \geq 1$ and $l \geq n$; furthermore, for any element of the domain

$$x = \begin{pmatrix} x_l & \dots & x_1 \end{pmatrix}$$

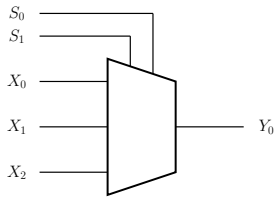
the image of this element is

$$c_n(x) = \begin{pmatrix} x_{l-n+1} & \dots & x_2 & x_1 \\ x_{l-n} & \dots & x_3 & x_2 \\ \vdots & & \vdots & \vdots \\ x_l & \dots & x_{n+1} & x_n \end{pmatrix}$$

For example in Figure 7.9 we show four input-output pairs for 2-carry.

## 7.4 Comparing our Three Algorithms

In this section we compare the performance of *blind_search_one*, *mutation_search_one* and *genetic_search_one*.

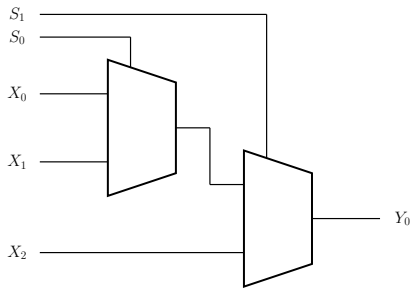



(a) A schematic of 3-multiplexer.

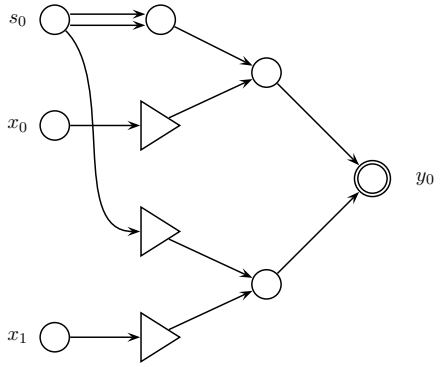
| $s_0$ | $s_1$ | $x_0$ | $x_1$ | $x_2$ | $y_0$ |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 1 | 1 |
| 0 | 0 | 1 | 1 | 0 | 1 |
| 0 | 0 | 1 | 0 | 1 | 1 |
| 0 | 0 | 1 | 0 | 0 | 1 |
| 0 | 0 | 0 | 1 | 1 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 | 1 |
| 1 | 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 | 1 |
| 0 | 1 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 | 0 | 0 |

(b) A truth table for 3-multiplexer.

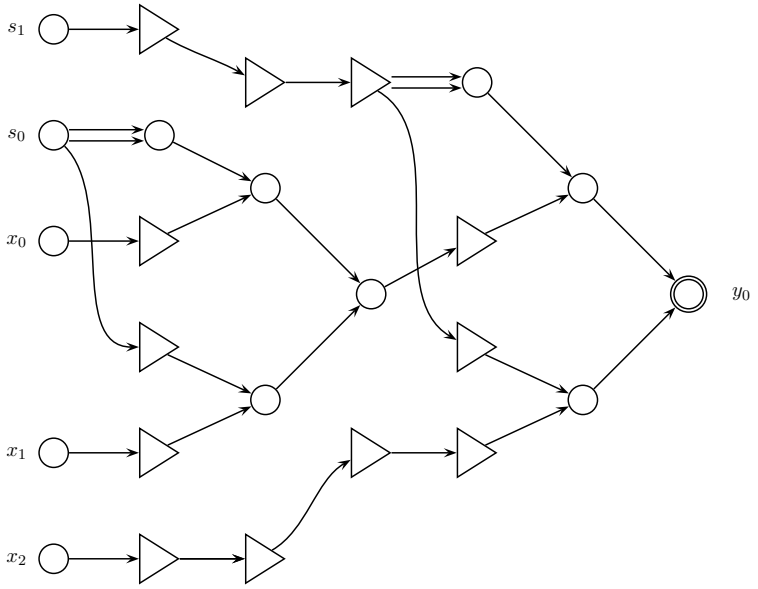
**Figure 7.7:** The 3-multiplexer.




(a) Schematically showing how a 3-multiplexer can be composed of two 2-mutliplexers.




(b) An A-type, with a delay $\delta = 3$, that represents columnwise 2-multiplexer.



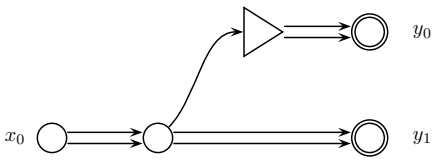

(c) An A-type, with a delay $\delta = 6$ that represents columnwise 3-multiplexer.

**Figure 7.8:** Constructing an A-type that represents columnwise 3-mutliplexer by composing two A-types that represent columnwise 2-multiplexer.

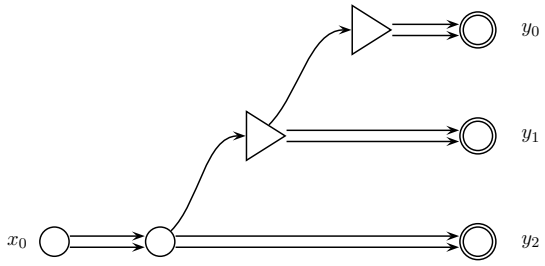
$$\left( \begin{array}{|c|c|c|} \hline 1 & 0 & 1 \\ \hline \end{array} \;,\; \begin{array}{|c|c|} \hline 0 & 1 \\ \hline 1 & 0 \\ \hline \end{array} \right), \qquad \left( \begin{array}{|c|c|c|} \hline 1 & 1 & 1 \\ \hline \end{array} \;,\; \begin{array}{|c|c|} \hline 1 & 1 \\ \hline 1 & 1 \\ \hline \end{array} \right),$$

$$\left( \begin{array}{|c|c|c|} \hline 1 & 0 & 0 \\ \hline \end{array} \;,\; \begin{array}{|c|c|} \hline 0 & 0 \\ \hline 1 & 0 \\ \hline \end{array} \right), \qquad \left( \begin{array}{|c|c|c|} \hline 0 & 0 & 1 \\ \hline \end{array} \;,\; \begin{array}{|c|c|} \hline 0 & 1 \\ \hline 0 & 0 \\ \hline \end{array} \right)$$

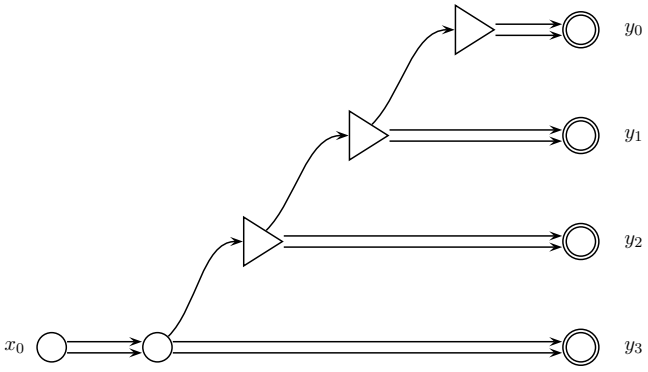
**Figure 7.9:** Four input-output pairs of 2-carry.




(a) An A-type, with a delay $\delta = 3$ that represents 2-carry.




(b) An A-type, with a delay $\delta = 4$ that represents 3-carry.




(c) An A-type, with a delay $\delta = 5$ that represents 4-carry.

**Figure 7.10:** A-types that represent $n$-carry for $n \in \{2, 3, 4\}$

### 7.4.1 Appropriate Training Data

In Section 7.2.2 we described our method of determining suitable training data for sequential A-types. We adopt this method here because it is sufficient when we compare our three algorithms: we require a training set that often results in a solution, rather than requiring an optimal training set.

Although all of our benchmark concepts can be used with sequential A-types we only do so with one of these concepts. When we use $n$-identity, $n$-parity, and $n$-multiplexer we do so with clamped A-types. This makes our investigations computationally easier. Conducting numerous trials with several $n$ values is very computationally expensive if we use sequential A-types. Given that we defined our A-types (with delay machines) so that they can operate sequentially, we chose to also benchmark our algorithms with a sequential task, namely $n$-carry.[†].

When we search for $n$-identity, $n$-parity, and $n$-multiplexer we do so with clamped A-types; consequently, we need suitable training data for clamped A-types. For example, let us consider the Boolean function 2-identity $id_2 : [\mathbb{Z}_2]^2 \rightarrow [\mathbb{Z}_2]^2$. Figure 7.11 illustrates an A-type $A$ that represents this function; Figure 7.12 demonstrates $A$'s response to all four possible inputs. For each of the four input packets the first column of bits generated after $\delta = 3$ moments is the output required by $id_2$, but for some input the response is oscillatory. That is, future columns of bits may vary—the *output* of $A$ is not clamped.

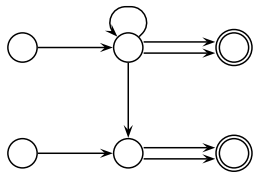


**Figure 7.11:** An A-type $A$, with delay $\delta = 3$, that represents the identity $id_2 : [\mathbb{Z}_2]^2 \rightarrow [\mathbb{Z}_2]^2$.

When searching for a clamped A-type that represents some concept $c$ an *exact solution* is an A-type that represents $c$ for every possible clamped example of $c$. Performing searches with long examples takes a long time; performing searches with short examples usually leads to inexact solutions. For all[‡] clamped searches described in this chapter that search for a benchmark concept $b$ the training set contains all possible examples of $b$ whose data packets are of length three. For example, Figure 7.13 shows the training set that we used when we searched for clamped 2-parity. We deem a solution A-type exact if it represents all clamped training examples of $b$ with data packets of length 1000. In each of our experiments most

[†]Note that because $n$-carry is the only benchmark concept that we search for with sequential A-types it is only in these searches that our A-types require delay machines. In spite of this, all of our searches use A-types with delay machines. This does not invalidate any of our comparisons of the three algorithms, but our unnecessary inclusion of delay machines may confuse the reader. When we performed our searches with clamped A-types it may have been clearer if we had employed A-types without delay machines.

[‡]We make a slight exception for $n$-identity; we detail this in Section 7.4.5.

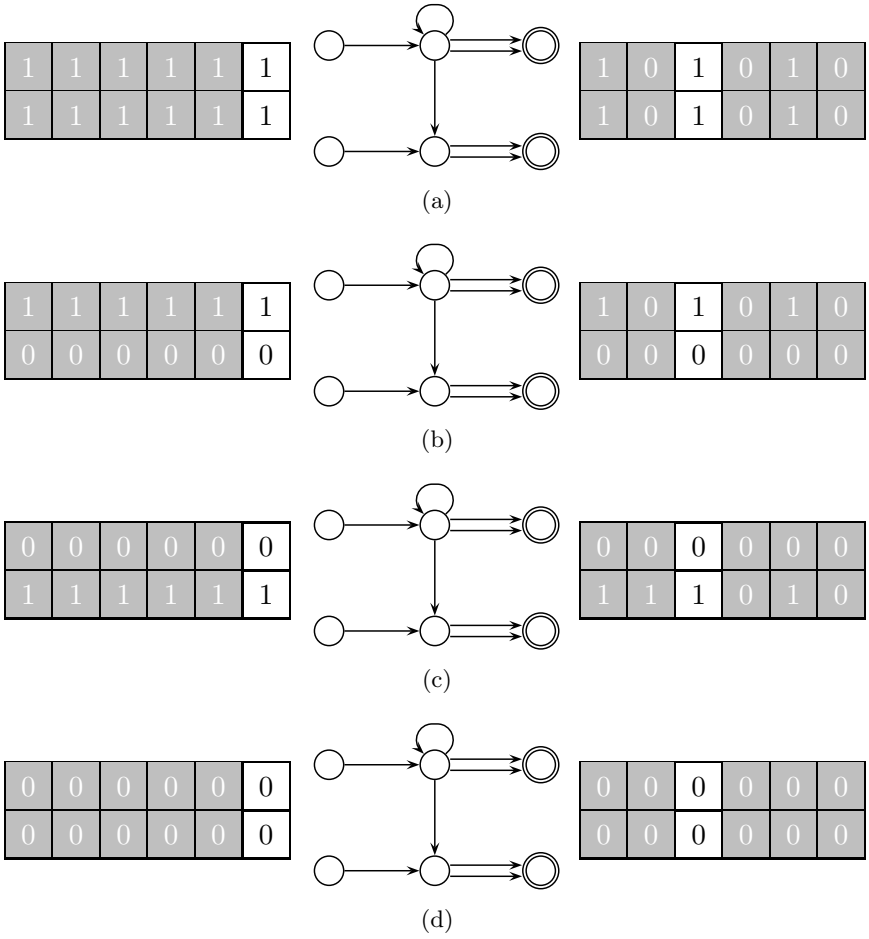



**Figure 7.12:** The A-type, with $\delta = 3$, shown in Figure 7.11 is a solution to the identity $id_2 : [\mathbb{Z}_2]^2 \to [\mathbb{Z}_2]^2$; however, it is not a clamped solution because the output is oscillatory in some of the generated data packets.

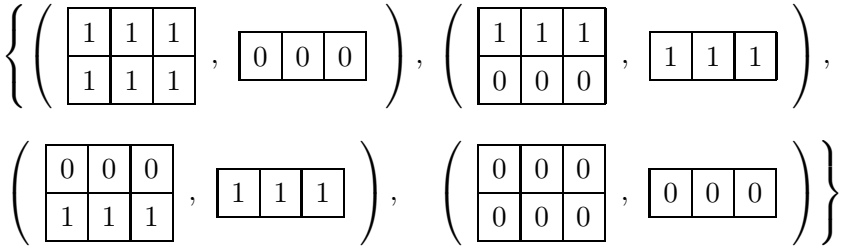


**Figure 7.13:** The training set that we used for our searches for 2-parity. Note that this set is exhaustive in that this set contains all possible examples of 2-parity that have data packets that have length three.

solutions returned were exact. Consequently, we believe that our training sets adequately represent the examples being learnt.

### 7.4.2 Other Search Parameters

For each of the three algorithms tested there are several parameters that require arguments; for instance, the population size, the selection rules, and the mutation to crossover ratio. To optimize each algorithm we need to search for appropriate arguments; furthermore, these arguments may be specific to each benchmark concept. We performed some rather informal investigations to decide upon arguments for these parameters. Those common to all four tests are presented in Table 7.5. Further details are presented as we introduce the investigations for each concept. There are two reasons why we were rather informal with our selection of arguments. First, establishing appropriate arguments is a substantial undertaking. We simply assume that the arguments that we have chosen suffice to compare our EAs to the blind search and to investigate the utility of our crossover operator. Second, the discovery of optimal arguments for our EAs is a special case of techniques that we had hoped to employ in this project. In particular, we want to apply evolutionary techniques to discover appropriate evolutionary search methods. Because this aspect of our research is incomplete we only offer a speculative discussion in Chapter 9—there we suggest that an appropriate direction for future research with A-types is the evolution of evolution operators.

### 7.4.3 Task Management

We conducted our investigations using many cores of numerous computers. Consequently, we had to minimize any bias that this may introduce into our results. For any trial that required a random selection of training examples the same set of training examples was used for all algorithms compared. For instance, when searching for $n$-carry we employ a single random example of length 50. When we employ each algorithm (from the set { *blind_search_one*, *mutation_search_one*, *genetic_search_one* }) the first trial has the same training data. To

| parameter | | argument |
|---|---|---|
| description | symbol | |
| population size | $N_{init}$ | 100 |
| penalty bound | $n_o$ | u |
| pressure gradient | $m$ | $\frac{1}{2}$ |
| worst fitness of a solution | $f_{worst}$ | 0.00 |
| probability that a node is constructed as a delay machine | $d$ | 20% |
| mutant selection rule | - | random |
| breeder selection rule | - | fitness-based |
| termination rule | - | fitness-based |
| crossover to mutation ratio for *genetic_search_one* (with crossover) | $n_{cross} : n_{muts}$ | 1 |
| maximum size (% parent's internal nodes) of donor or acceptor subgraphs for crossover. | $p_{max}$ | 80% |

**Table 7.5:** Parameters common to all to four investigations in this section. Note that the penalty bound $u$ is the largest A-type for each initial population. This is specific to each investigation.

measure CPU time we employed Java's ThreadMXBean interface[§]. When we compared our CPU times with those of linux's *time* command (when we summed the system and user times) we found a constant difference of approximately two seconds. Given the long run times for most of our trials we are confident that our measure of elapsed CPU time is reasonable. Our count of attempts performed (that is the number of A-types constructed either in the initial population, via mutation, or via crossover) is independent of the computer that the test is conducted on. This gives us confidence that when we distribute our trials over many computers we can achieve an accurate comparison of our algorithms.

### 7.4.4 Uncertainties

For all of the results that we present in this section we employ Student's $t$-test (for instance see [82, sec 24.6]) to determine a confidence interval. We chose to display 90% confidence intervals. That is, when we plot our results we display the mean within error bars that span the interval that we are 90% confident contains the true mean. We assume that our results are normally distributed for the $t$-test to be valid.

[§]Specifically, we used the method ThreadMXBean.getCurrentThreadCpuTime() [81]

| parameter | | argument |
|---|---|---|
| description | symbol | |
| smallest size of initial machine | $l$ | $3n$ |
| largest size of initial machine | $u$ | $4n$ |
| max num of attempts | $N_{births}$ | $10^7$ |
| trials per training example | - | 30 |
| length of exact solution | - | $10^3$ |

**Table 7.6:** Parameters used for our clamped $n$-identity searches.

### 7.4.5 Searching for Clamped $n$-identity

In this section we search for A-types that represent clamped $n$-identity for values of $n$ that range from one to ten. In Table 7.6 we list arguments that we chose for this search. When we search for clamped $n$-identity we employ the maximum number of examples whose data packets are of length three, unless this maximum exceeds 100 (this is when $n \in \{7, 8, 9, 10\}$). In the latter case we randomly choose 100 examples, whose data packets are of length three, for each trail. However, each algorithm is tested with the same set of sets of training data. For instance, when searching for 8-identity for each $i$ the $i$th trial with *blind_search_one*, the $i$th trial with *mutation_search_one*, and the $i$th trial with *genetic_search_one* all use exactly the same examples. Our choice of training data almost always gave exact solutions and, as described above, we ensured that it was not a variable when we compared our algorithms.

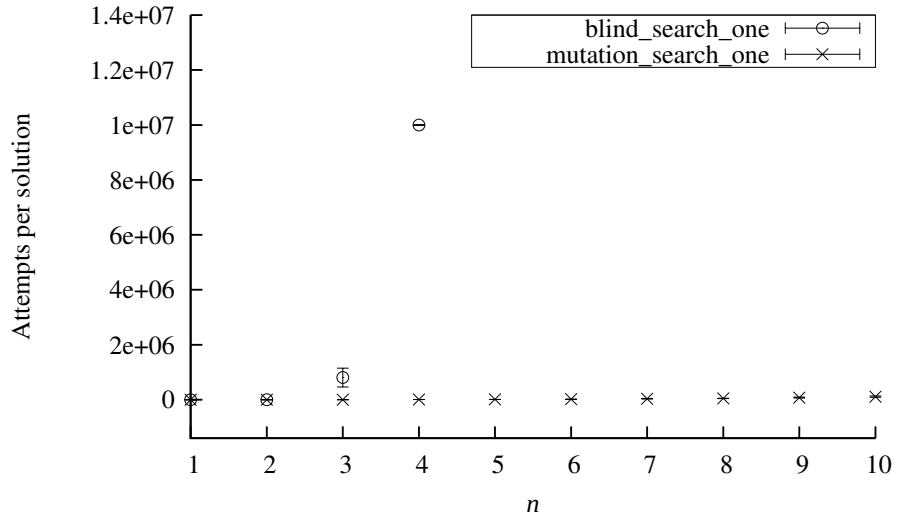
In Figure 7.14 we compare the performance of *blind_search_one* to *mutation_search_one*. We do not display results for *blind_search_one* for $n > 4$. This is because using *blind_search_one* to search for clamped $n$-identity yielded no solutions for $n > 4$. These results show that *mutation_search_one* outperforms *blind_search_one* by orders of magnitude both in terms of processing time and in terms of required number of attempts.

In Figure 7.15 we compare the performance of *mutation_search_one* to *genetic_search_one*. These results show that *genetic_search_one* significantly outperforms *mutation_search_one* both in terms of processing time and in terms of required number of attempts. This result provides evidence that problems exist whose solution may be discovered more efficiently when we include our crossover operator.

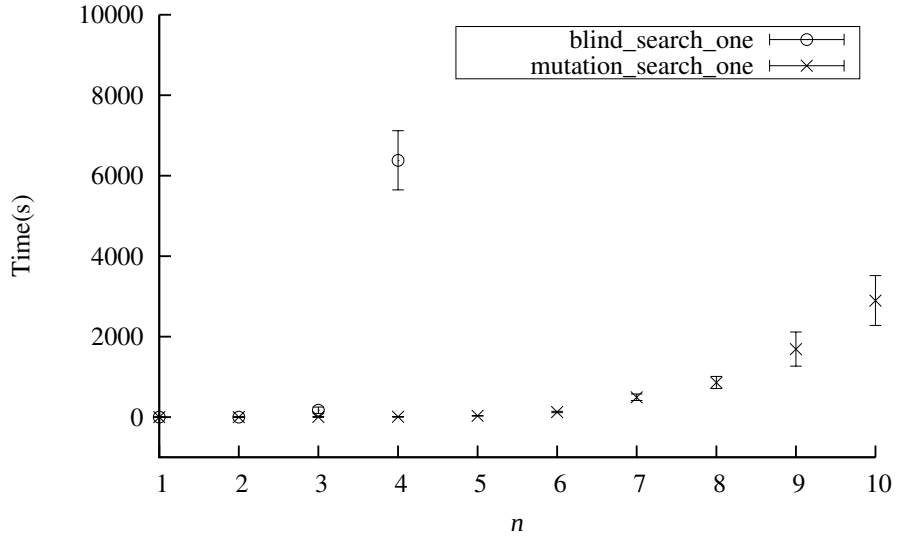
### 7.4.6 Searching for Clamped $n$-parity

In this section we search for clamped A-types that represent $n$-parity for values of $n$ that range from two to five. In Table 7.7 we list arguments that we chose for this search.

In Figure 7.16 we compare the performance of *blind_search_one* to *mutation_search_one*. For $n = 3$ *blind_search_one* failed to find any solution for all trials. We include the data

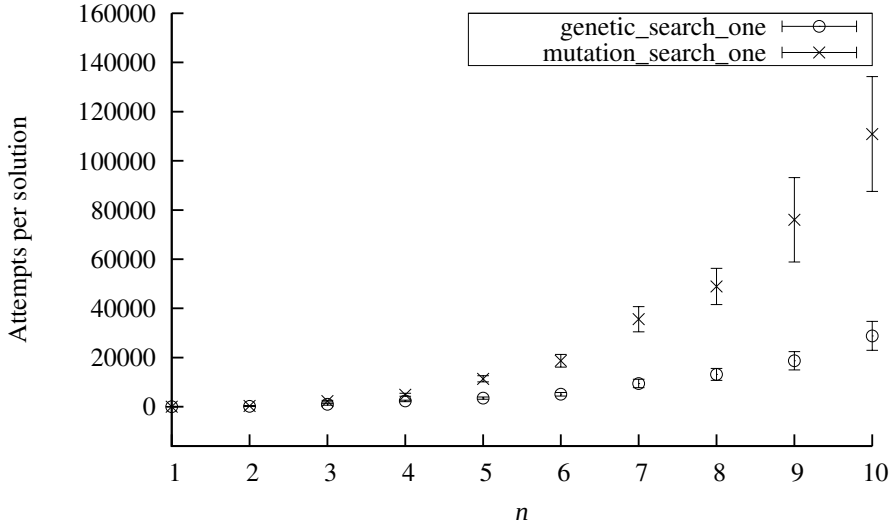



(a) The average number of attempts required before a solution was discovered.

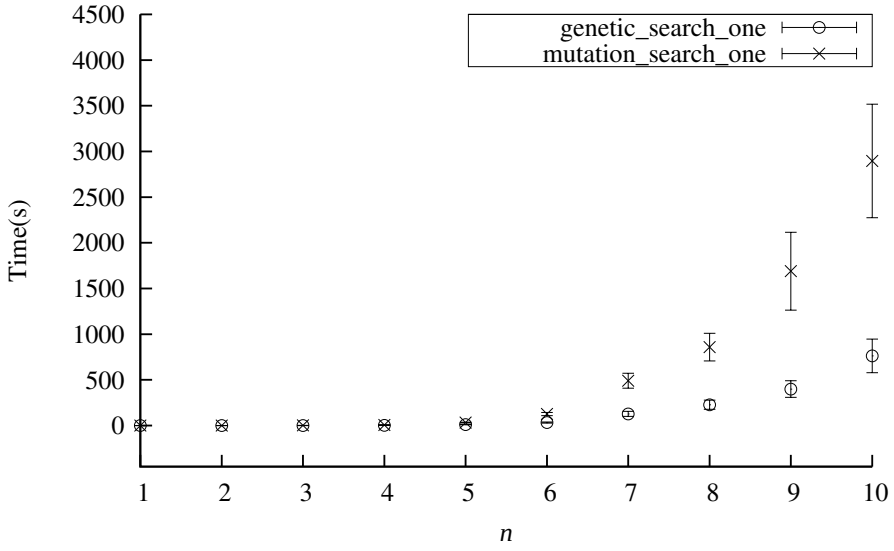



(b) The average CPU time required before a solution was discovered.

**Figure 7.14:** *n-identity:* Comparing the performance of *blind_search_one* and *mutation_search_one* when searching for clamped $n$-identity with $n$ ranging from 1 to 10. Note that for $n > 4$ all trials using *blind_search_one* failed to find a solution within $10^7$ attempts.




(a) The average number of attempts required before a solution was discovered.




(b) The average CPU time required before a solution was discovered.

**Figure 7.15:** *n-identity:* Comparing the performance of *mutation_search_one* and *genetic_search_one* when searching for clamped $n$-identity with $n$ ranging from 1 to 10.

| parameter | | argument |
|---|---|---|
| description | symbol | |
| smallest size of initial machine | $l$ | $5(n-1)$ |
| largest size of initial machine | $u$ | $6(n-1)$ |
| max num of attempts | $N_{births}$ | $10^8$ |
| trials per training example | - | 30 |
| length of exact solution | - | $10^3$ |

**Table 7.7:** Parameters used for our clamped $n$-parity searches.

| parameter | | argument |
|---|---|---|
| description | symbol | |
| smallest size of initial machine | $l$ | † |
| largest size of initial machine | $u$ | $l+4$ |
| max num of attempts | $N_{births}$ | $10^8$ |
| trials per training example | - | 20 |
| length of exact solution | - | $10^3$ |

**Table 7.8:** Parameters used for our clamped $n$-multiplexer searches. † Note that for the lower bound we devised the following function $l$ of $n$. $\{l(2) = 7, l(3) = 13, l(4) = 18, l(5) = 24\}$ by examining solutions constructed composing copies of our 2-multiplexer.

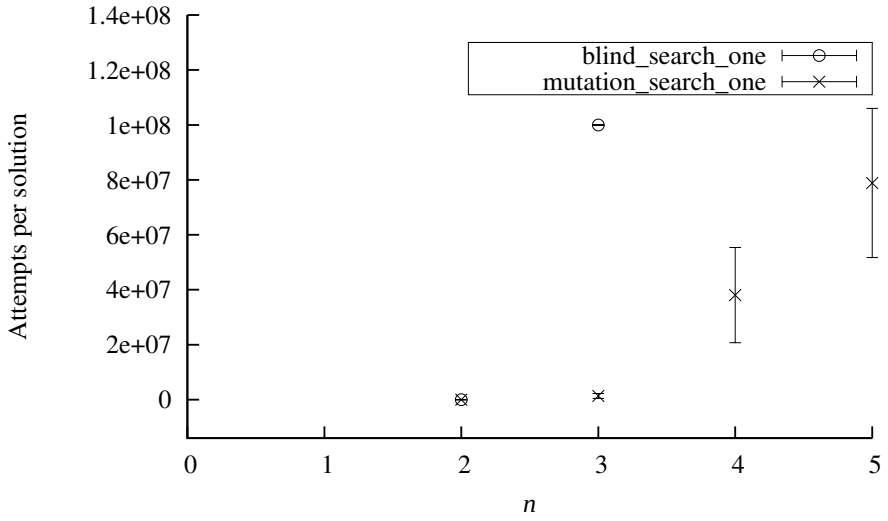
points that correspond to $n = 3$ as lower bounds for 3-parity on the two plots. These results show that *mutation_search_one* significantly outperforms *blind_search_one* both in terms of processing time and in terms of required number of attempts.

In Figure 7.17 we compare the performance of *mutation_search_one* to *genetic_search_one*. These results show that when searching for $n$-parity for $n \in \{2, 3, 4, 5\}$ there is no difference between the performance of our two EAs.

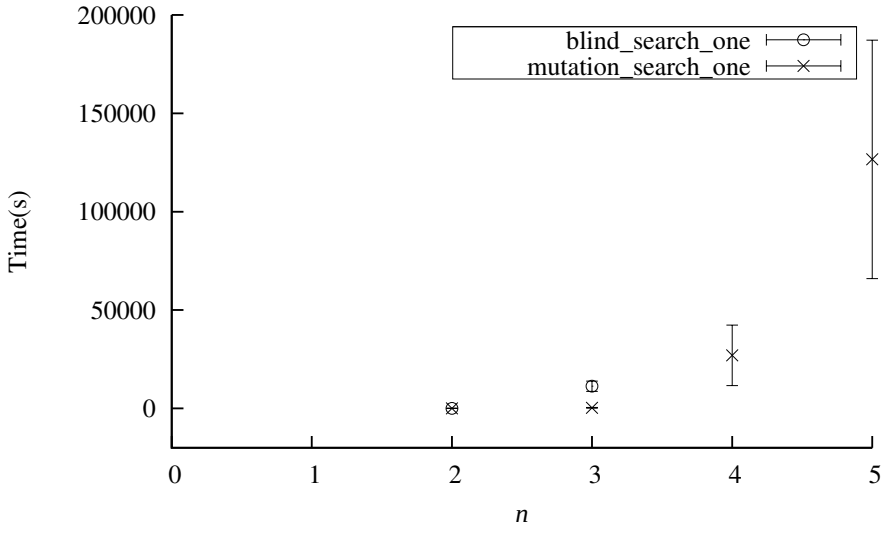
### 7.4.7 Searching for Clamped $n$-multiplexer

In this section we search for clamped A-types that represent $n$-multiplexer for values of $n$ that range from two to five. In Table 7.8 we list arguments that we chose for this search.

In Figure 7.18 we compare the performance of *blind_search_one* to *mutation_search_one*. When we used *blind_search_one* to search for A-types that represented 3-multiplexer only two of the twenty trials returned a solution (before $10^8$ attempts). We include the data point corresponding to $n = 3$ for *blind_search_one* as a lower bound. These results show that *mutation_search_one* significantly out-performs *blind_search_one* both in terms of processing time and in terms of required number of attempts.

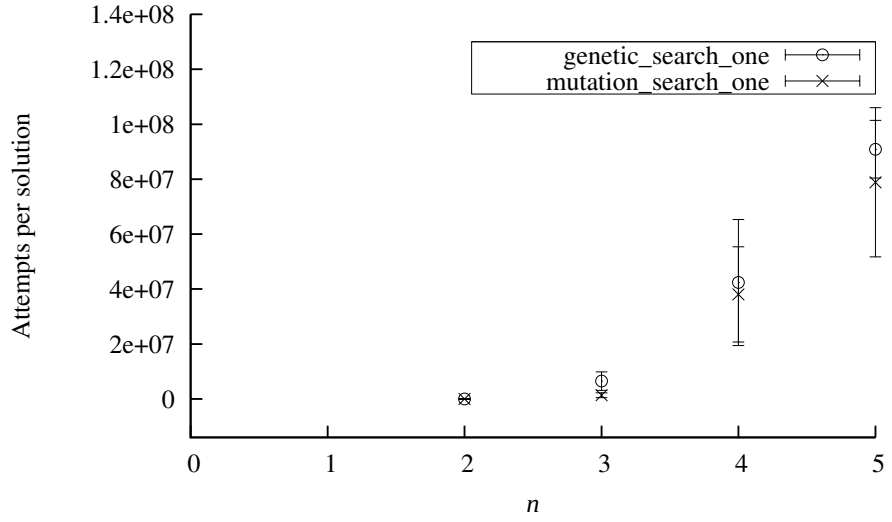



(a) The average number of attempts required before a solution was discovered.

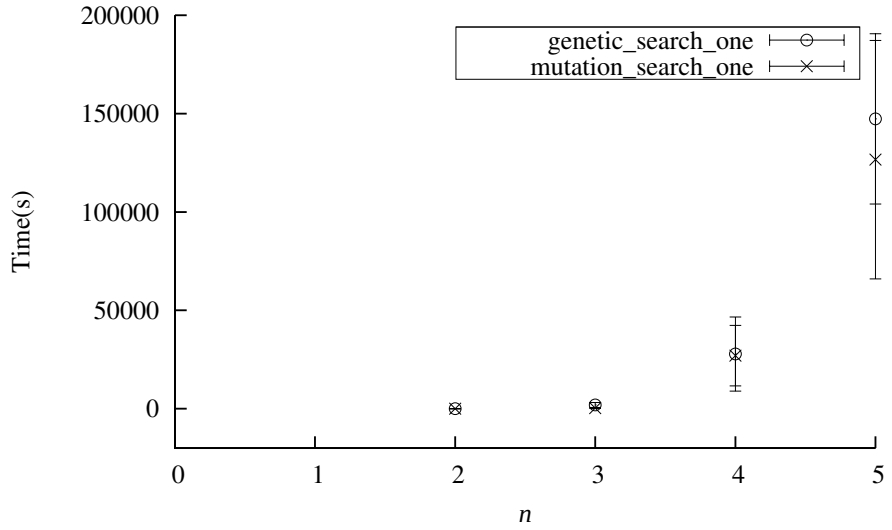



(b) The average CPU time required before a solution was discovered.

**Figure 7.16:** *n-parity:* Comparing the performance of *blind_search_one* and *mutation_search_one* when searching for clamped $n$-parity with $n$ ranging from 2 to 5. Note that for $n = 3$ using *blind_search_one* all trials failed to find a solution; so, the corresponding data points displayed are lower bounds.



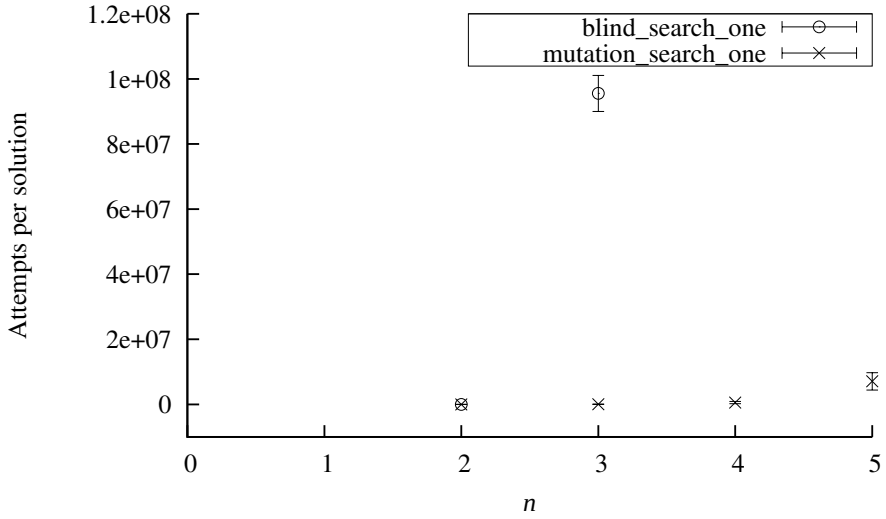

(a) The average number of attempts required before a solution was discovered.



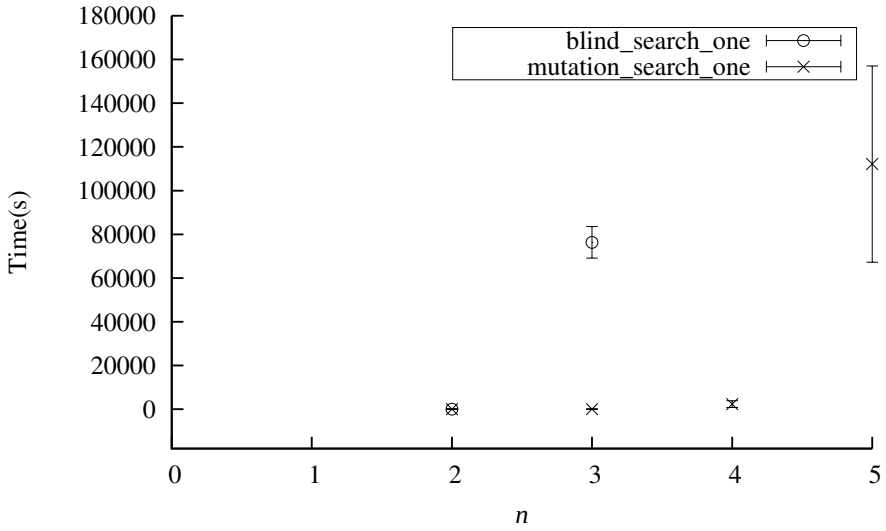

(b) The average CPU time required before a solution was discovered.

**Figure 7.17:** *n-parity:* Comparing the performance of *mutation_search_one* and *genetic_search_one* when searching for clamped $n$-parity with $n$ ranging from 2 to 5.

In Figure 7.19 we compare the performance of *mutation_search_one* to *genetic_search_one*. These results show that when searching for $n$-multiplexers for $n \in \{2, 3, 4, 5\}$ there is no difference between the performance of our two EAs.

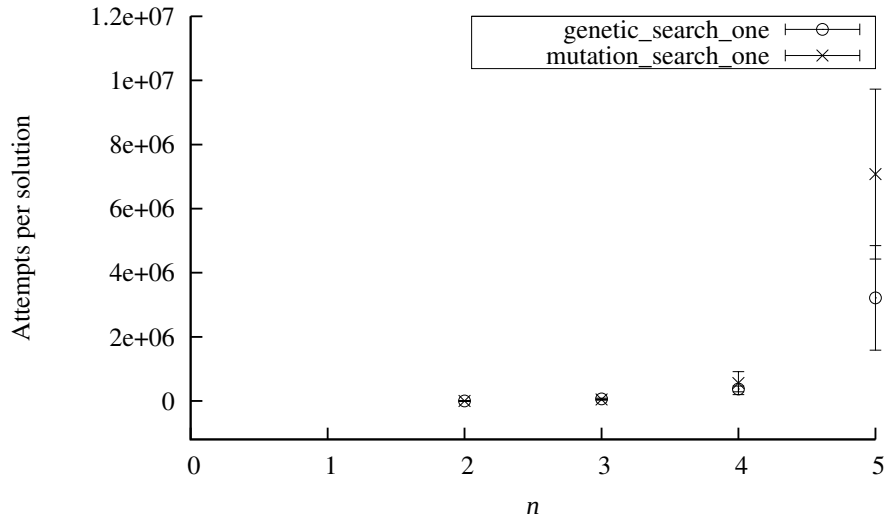



(a) The average number of attempts required before a solution was discovered.

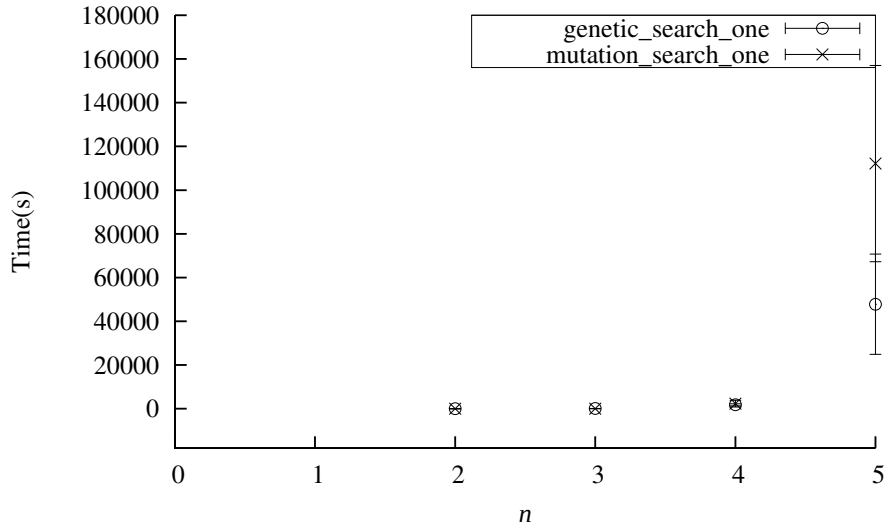



(b) The average CPU time required before a solution was discovered.

**Figure 7.18:** *n-multiplexer:* Comparing the performance of *blind_search_one* and *mutation_search_one* when searching for clamped $n$-multiplexer with $n$ ranging from 2 to 5. Note that the point corresponding to $n = 3$ for *blind_search_one* is a lower bound.




(a) The average number of attempts required before a solution was discovered.




(b) The average CPU time required before a solution was discovered.

**Figure 7.19:** *n-multiplexer:* Comparing the performance of *mutation_search_one* and *genetic_search_one* when searching for clamped $n$-multiplexer with $n$ ranging from 2 to 5.

| parameter | | argument |
|---|---|---|
| description | symbol | |
| smallest size of initial machine | $l$ | $3 + 2(n-1)$ |
| largest size of initial machine | $u$ | $3 + 2n$ |
| Max num of attempts | $N_{births}$ | $10^8$ |
| trials per training example | - | 20 |
| length of exact solution | - | $10^4$ |

**Table 7.9:** Parameters used for our clamped $n$-carry searches.

### 7.4.8 Searching for $n$-carry (Sequential)

In this section we search for clamped A-types that represent $n$-carry for values of $n$ that range from 2 to 7. In Table 7.9 we list arguments that we chose for this search.

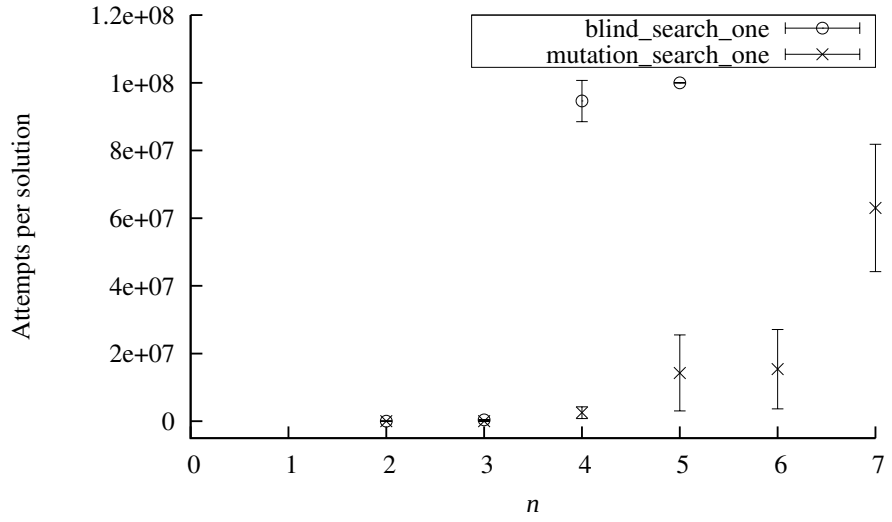
In Figure 7.20 we compare the performance of *blind_search_one* to *mutation_search_one*. Note that using *blind_search_one* all trials for $n > 4$ failed to find a solution. We include the points corresponding to *blind_search_one*'s 5-carry results as a lower bound. From Figure 7.20 we see that *mutation_search_one* significantly outperforms *blind_search_one* both in terms of processing time and in terms of required number of attempts.

In Figure 7.21 we compare the performance of *mutation_search_one* to *genetic_search_one*. These results show that *genetic_search_one* significantly outperforms *mutation_search_one* both in terms of processing time and in terms of required number of attempts.

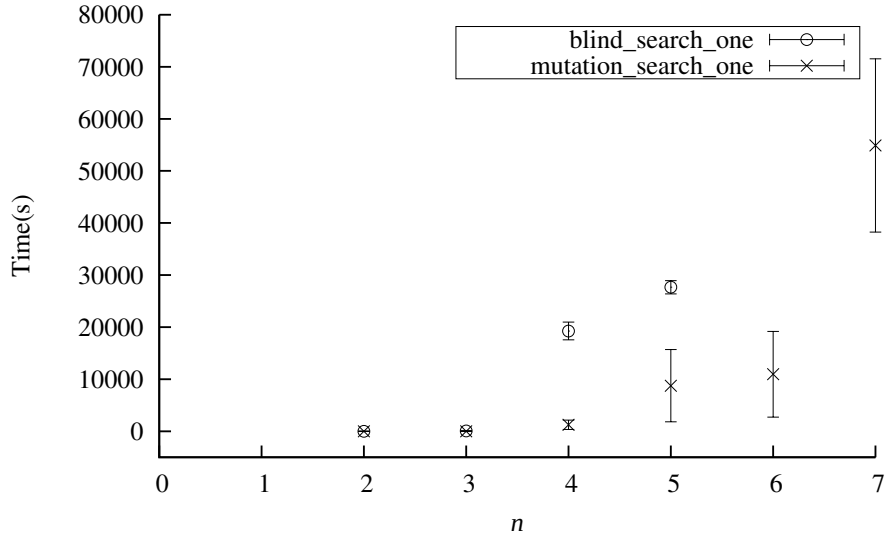
## 7.5 Is our Crossover Simply Macromutation

We invested substantial effort designing, and implementing the crossover operator for *genetic_search_one* (see Section 6.6). With the results from the $n$-identity searches and the $n$-carry searches we demonstrate that in our EA, for some tasks, our crossover is useful. In many EAs crossover is useful because it provides sudden large variation in the population, rather than because it recombines individuals [43, ch 6]. Such an operator is called a macromutation operator. As we noted in Section 3.4.1 the utility of *biological* crossover is due to its ability to recombine individuals' information.

When investigating whether an EA's crossover is simply a macromutation operator the 'headless chicken' search offers a relatively simple means testing whether a crossover operator is simply acting as a macromutator [83] [84]. The headless chicken search is an EA where only one parent is selected from the population and the other parent is an entirely new individual [43, p153]. We implemented the headless chicken algorithm by duplicating *genetic_search_one* with the following modification. For each crossover, after we have selected the parents ♀, ♂ we randomly choose one parent $P$ and then construct a random A-type $P'$

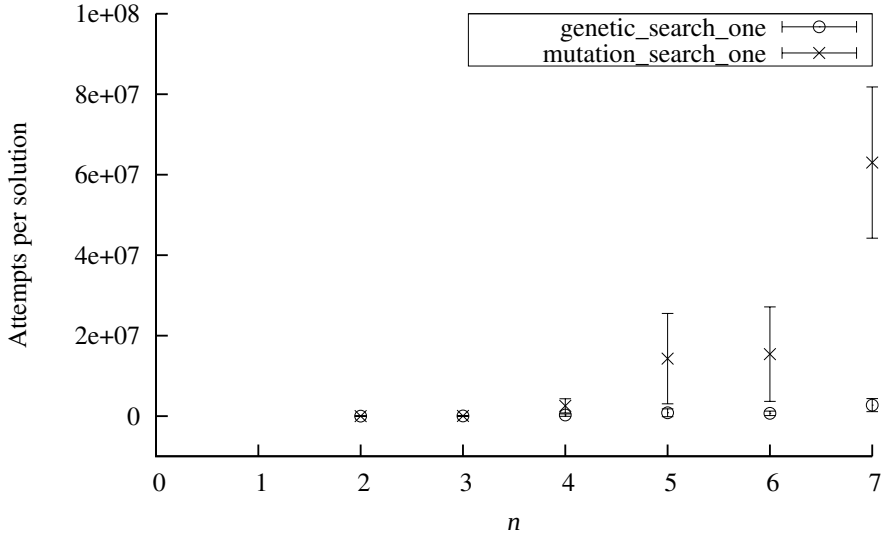



(a) The average number of attempts required before a solution was discovered.

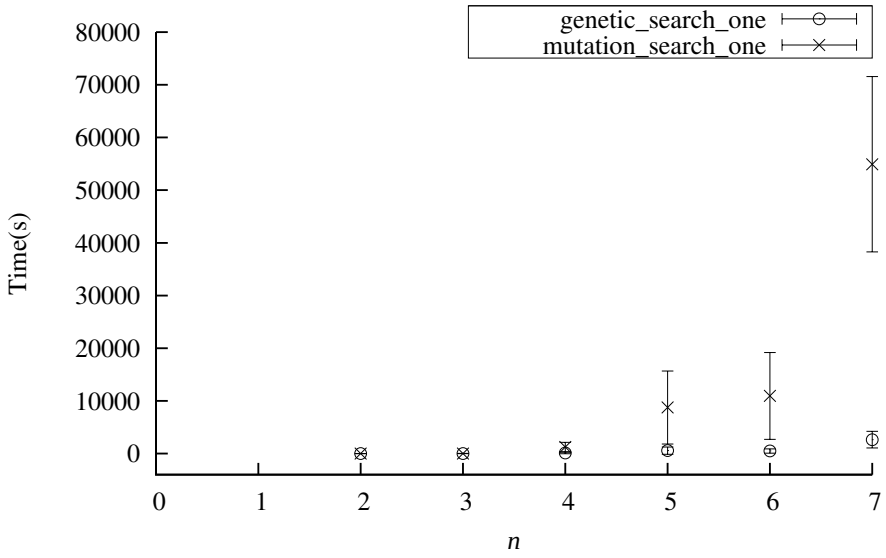



(b) The average CPU time required before a solution was discovered.

**Figure 7.20:** *n-carry:* Comparing the performance of *blind_search_one* and *mutation_search_one* when searching for $n$-carry with $n$ ranging from 2 to 7. Note that the data points corresponding to $n = 5$ for *blind_search_one* are lower bounds only.




(a) The average number of attempts required before a solution was discovered.




(b) The average CPU time required before a solution was discovered.

**Figure 7.21:** *n-carry:* Comparing the performance of *mutation_search_one* and *genetic_search_one* when searching for $n$-carry with $n$ ranging from 2 to 7.

that is the same size as $P$. If $P$ was ♀ then we perform the crossover using $P'$ and ♂; if $P$ was ♂ then we perform the crossover using ♀ and $P'$.

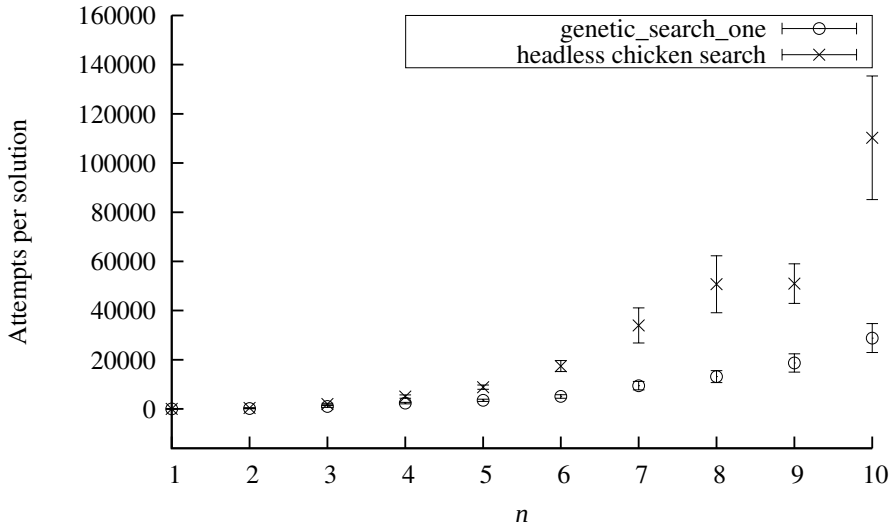
We compare *genetic_search_one* and our headless chicken search on the benchmark tasks that demonstrated the utility of our crossover: when searching for $n$-identity and when searching for $n$-carry. Figure 7.22 shows that when searching for $n$-identity *genetic_search_one* outperforms our headless chicken search. Figure 7.23 shows that when searching for $n$-carry *genetic_search_one* outperforms our headless chicken search.

This shows that, for some tasks, our crossover operator, *crossover_one*, is more useful than a macromutation operator.

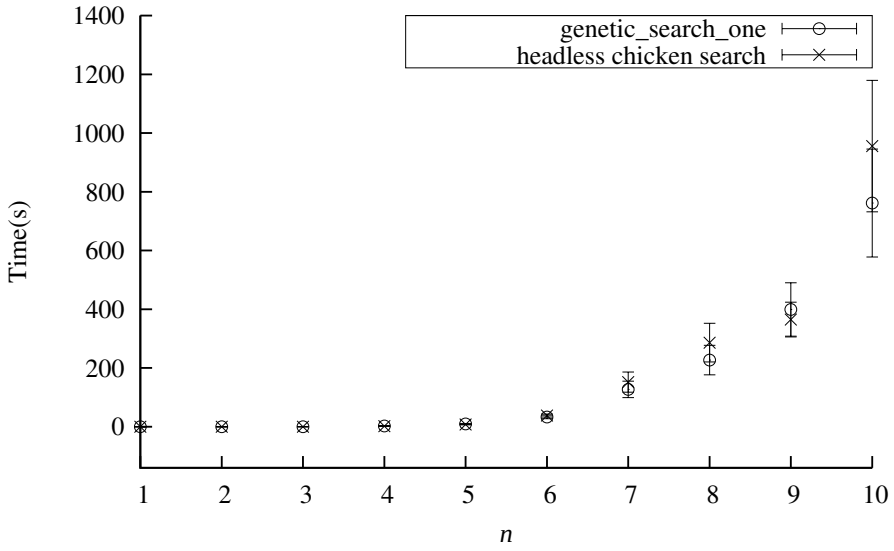
## 7.6 Conclusions

In this chapter we have achieved the following.

- ⋆ Provided evidence in support of Claim 5.3 and Claim 5.4. That is, we provide evidence that we require delay machines for our sequential A-types.
- ⋆ Provided four benchmark tests for which our EAs significantly outperformed the blind search.
- ⋆ Showed that when searching for $n$-identity and $n$-carry *genetic_search_one* outperformed *mutation_search_one*.
- ⋆ Demonstrated that, for some tasks, our crossover operator is more useful than a macromutation operator. We did this by conducting a headless chicken searches for $n$-identity and $n$-carry.

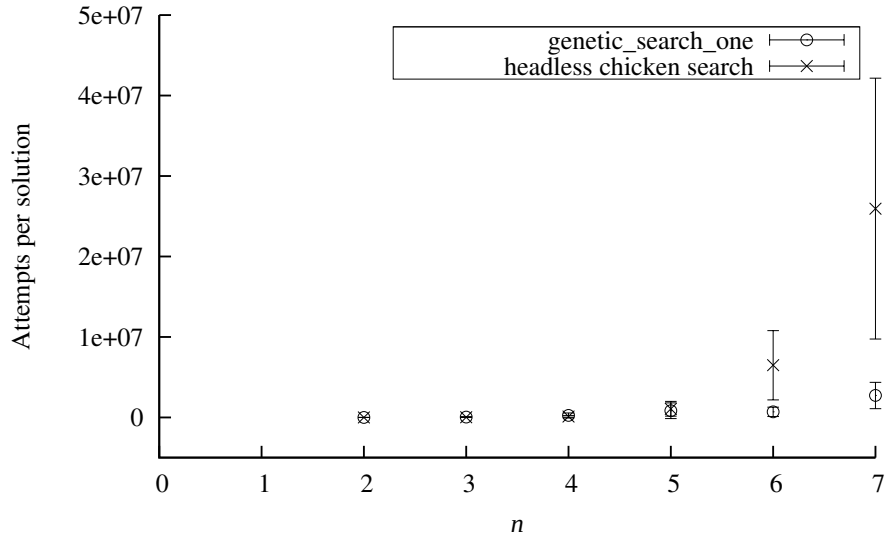



(a) The average number of attempts required before a solution was discovered.

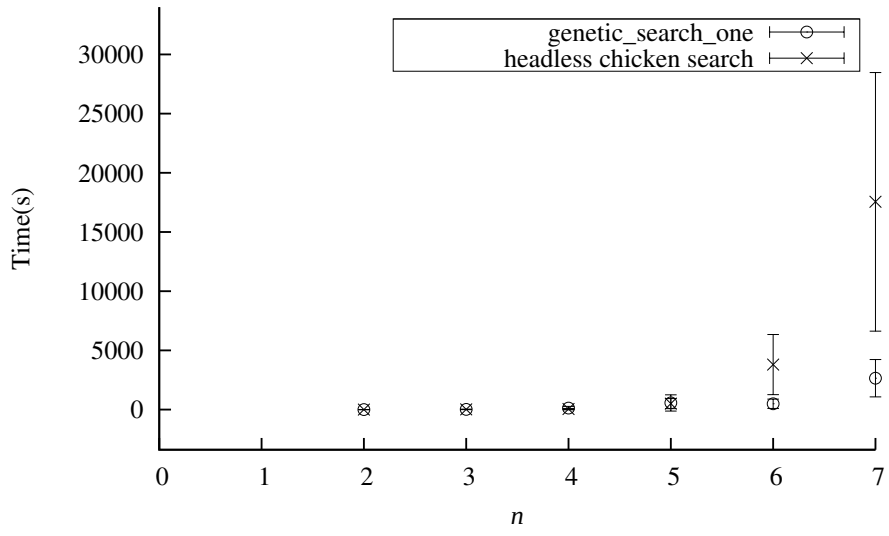



(b) The average CPU time required before a solution was discovered.

**Figure 7.22:** *n-identity (headless chicken):* Comparing the performance of *genetic_search_one* and *genetic_search_one* with a headless chicken crossover operator. Here we compare these algorithms as they search for clamped $n$-identity for $n$ ranging from 1 to 10.




(a) The average number of attempts required before a solution was discovered.




(b) The average CPU time required before a solution was discovered.

**Figure 7.23:** *n-carry (headless chicken):* Comparing the performance of *genetic_search_one* and *genetic_search_one* with a headless chicken crossover operator. Here we compare these algorithms as they search for clamped $n$-carry for $n$ ranging from 2 to 7.

# 8 Employing Ideas of Symmetry

## 8.1 Aims of this Chapter

In this chapter we present two techniques that attempt to improve our EAs with A-types when they search for a class of 'symmetric' functions. More precisely, we employ group theoretic ideas for EAs that search for solutions that are invariant under permutations of the input variables. These techniques attempt to decrease the hypothesis space with the aim of improving the search.

## 8.2 Symmetric Functions

In this section we make the notion of 'symmetric function' precise. Specifically, we only consider functions that map data packets to data packets. We employ the definition of a group invariant function (Definition 2.8) to define functions that are unaffected by permuting rows in the input data packets. First, let us define a particular group action (see Definition 2.2.1) that permutes rows of a $k \times l$ matrix, where $k, l \in \mathbb{Z}$. Consider the permutation group $S_l$. Let us define the function $\rho_{row} : S_l \times M[k,l] \to M[k,l]$ that maps every pair $(s, m)$ to a matrix $m'$ such that $m'$ is the result of permuting the rows of $m$ in the obvious way specified by $s$; for instance, under the permutation (12) the first and second rows are transposed. The function $\rho_{row}$ is an action of $S_l$ on $M[k,l]$. Second, consider a function $f : M[k,l] \to M[p,q]$ that maps data packets to data packets. Also consider a subgroup $B$ of the permutation group $S_l$. We say that $f$ is $B$-invariant if it is $B$-invariant with respect to $\rho_{row}$. That is, $f$ is $B$-invariant if, for any $b \in B$ the output of $f$ is unaffected when $b$ permutes the rows of the input of $f$.

Now we define the group invariance of an A-type. Consider an A-type $A$ with an input dimension $l$, and consider a group $B$ that is a subgroup of $S_l$. Also, consider a data packet $D$ that has $l$ rows and the set $\{\rho_{row}(D)\}$ of all possible data packets constructed by row permuting $D$ under elements of $B$. Finally, consider the set $\{D_{out}\}$ of all output data packets generated by $A$ with inputs from $\{\rho_{row}(D)\}$. The A-type $A$ is $B$-invariant if $\{D_{out}\}$ has exactly one element.

We now propose the following hypothesis.

**Claim 8.1.** Consider a group $B$ that is a subgroup of $S_l$. Also, consider some $B$-invariant function $f : M[k,l] \to M[p,q]$, where $k, l, p, q \in \mathbb{Z}^+$. We can devise an EA to search for

A-types that represent $f$ that uses $B$ to decrease its hypothesis space. Furthermore, this EA will perform better than any of the algorithms present in Chapter 6.

□

We implement two algorithms in an attempt to confirm the above hypothesis. Both algorithms are mutation-only EAs. We label the first scheme a top-down approach. It is *mutation_search_one* with a new fitness function. The new fitness function estimates each candidate solution A-type's group invariance to help determine that A-type's fitness. We label the second scheme a bottom-up approach: when we generate A-type candidate solutions we do it in such a way that they have the symmetry that we want.

## 8.3 Top-down Approach

Our first approach to verifying Claim 8.1 is to implement *mutation_search_one* with a modified fitness function. When searching for some concept $f$ that is $B$-invariant (for some subgroup $B$ of $S_l$ where $l$ is the number of rows of all $f$'s input data packets) we make the fitness of each candidate solution $A$ depend on an estimate of $A$'s $B$-invariance. We hypothesize that 'pressuring' a population to be 'more $B$-invariant' is advantageous. We call this scheme the *invariance_estimate* algorithm.

To estimate a candidate solution's group invariance we record the difference between outputs in response to permutations of a given input. For a sequential A-type we create a set of input data packets as follows.

1. Construct a long random data packet $D$.
2. Construct the set $\{\rho_{row}(D)\}$ of all possible data packets constructed by row permuting $D$ under elements of $B$.
3. Construct the set $\{D_{out}\}$ of all output data packets generated when $A$ accepts elements of $\{\rho_{row}(D)\}$ as input.
4. Measure the difference between all output data packets. That is, determine the average normalized Hamming distance between all pairs $(D_{out}, D'_{out})$ where $D_{out}$ and $D'_{out}$ are elements of $\{D_{out}\}$.

We tested the *invariance_estimate* algorithm by searching for clamped $n$-parity with the knowledge that $n$-parity is $S_n$-invariant. Consequently, we implemented a special case of the above method of measuring a candidate solution's group invariance. We describe this method in Table 8.1.

In the *invariance_estimate* algorithm we ensure that a candidate solution's fitness also depends on it's performance with respect to the training data. The fitness of each candidate solution is a linear combination of the fitness returned by $fitness\_one$ (see Section 6.4.3) and

**test_invariance**($A$) is an algorithm that, given an A-type A whose input dimension is $l$, returns an estimate of $A$'s $S_l$-invariance.

| Parameters | | |
|---|---|---|
| *Type* | *Parameter* | *Description* |
| A-type | $A$ | The A-type input dimension is $l$ whose $S_l$-invariance is estimated. |

The algorithm has the following steps

*For data packets of length $k$ and height $l$ let $D_{ri}$ denote the data packet that has every entry in the $i$th row assigned unity and all other entries assigned zero.*

1. Iterate through $(D_{ri}, D_{rj})$ pairs and maintain a running average of the Hamming distance between the outputs generated by $A$ with $D_{ri}$ and $D_{rj}$ as input.

   for $i \in \{1, l\}$

   for $j \in \{(i+1), l\}$

   A. Determine $A$'s output given each of the current $D_{ri}$ and $D_{rj}$.

   B. Determine the normalized Hamming distance between these two outputs.

   C. Add this current normalized Hamming distance to the running average.

2. Return the average normalized Hamming distance

**Table 8.1:** Our method for estimating the $S_n$ invariance of a *clamped* A-type that has $n$ input nodes.

our estimate of the candidate solution's group invariance. We call our new fitness function $fitness\_two$ and we describe it in Table 8.2.

### 8.3.1 Testing the invariance_estimate Algorithm

In this section we present a comparison of *invariance_estimate* to *genetic_search_one*. Both algorithms were employed to search for $n$-parity with $n \in \{2, 3, 4, 5\}$. The results for our searches with *genetic_search_one* were presented in Section 7.4.6. In that section we specified arguments chosen for parameters of the searches. These arguments were also used when we tested the *invariance_estimate* EA, with one exception: when searching with *invariance_estimate* we only performed 10 trials for each $n$. We chose this smaller number of trials because the trials took a considerable time to complete. The results are shown in Figure 8.1. The results show no difference in the performance of the two algorithms. We also examined the performance of *invariance_estimate* using three different arguments for

**fitness_two**$(A, sym\_weight)$ is a fitness function that incorporates an estimate of an A-type's group invariance.

| Parameters | | |
|---|---|---|
| *Type* | *Parameter* | *Description* |
| A-type | $A$ | The A-type whose input dimension is $l$ and whose $S_l$-invariance is estimated. |
| $[0, 1]$ | *symm_weight* | A real number that weights the influence of our estimate of $A$'s group invariance. |

The algorithm has the following steps

1. Assess $A$'s fitness with respect to the search's training data.
   $x_T \leftarrow fitness\_one(A)$.
2. Assess $A$'s $B$-invariance.
   $x_B \leftarrow test\_invariance(A)$.
3. Use $Symm\_weight$ to determine a linear combination of the above results.
   $fitness \leftarrow (sym\_weight)(x_B) + (1 - sym\_weight)(x_T)$.
4. return $fitness$

**Table 8.2:** The fitness function that is used in the *invariance_estimate* algorithm

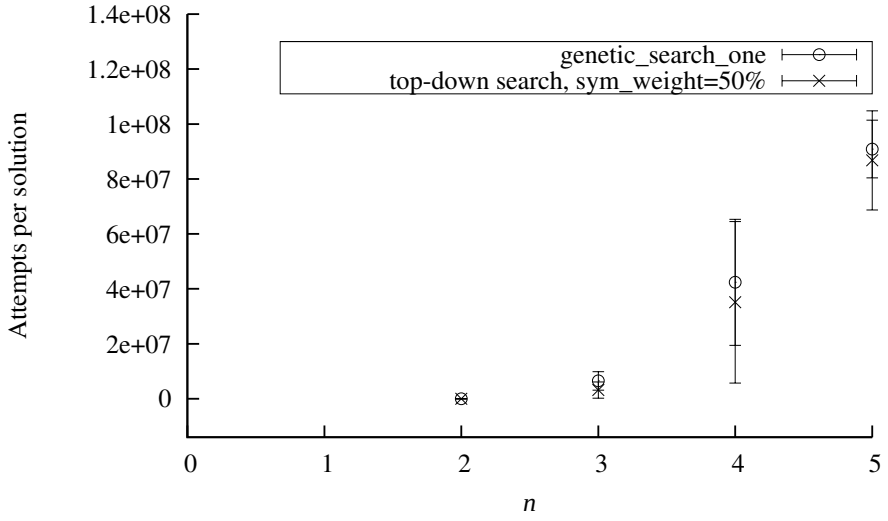
*sym_weight* as it searched for $n$-parity. We display the results of this investigation in Figure 8.2. The trials performed with $sym\_weight = 25\%$ are favourable in comparison to the other two searches. In light of this result it is possible that *invariance_estimate* may be optimized to be a useful search method. In conclusion, this requires further investigation.

## 8.4 A Bottom-Up Approach

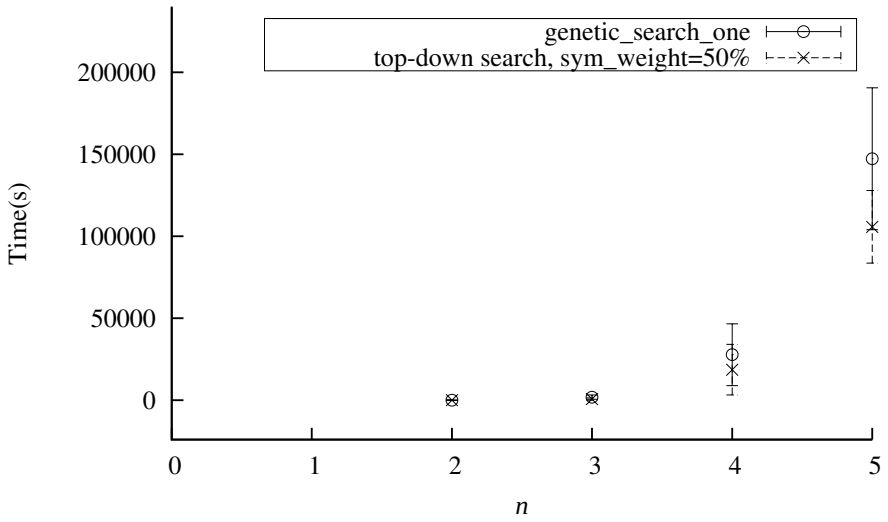
In this section we detail our second, more complex, attempt at verifying Claim 8.1. We classify this approach as a bottom-up method because all of our candidate solutions have an appropriate symmetry. This scheme is an EA the hypothesis space of which is populated with (A-type, automorphism) pairs, where the automorphism encodes the A-type's symmetry*.

Consider the A-type $A$ shown in Figure 8.3(a). This A-type (with a delay of three) is

*Shaw-Taylor [14] also employs (ANN, automorphism) pairs to encode the symmetry of a problem into an ANN. Our method diverges from Shaw-Taylor's because our implementation permits the evolution of populations of (ANN, automorphism) pairs. Furthermore, our networks allow feedback.

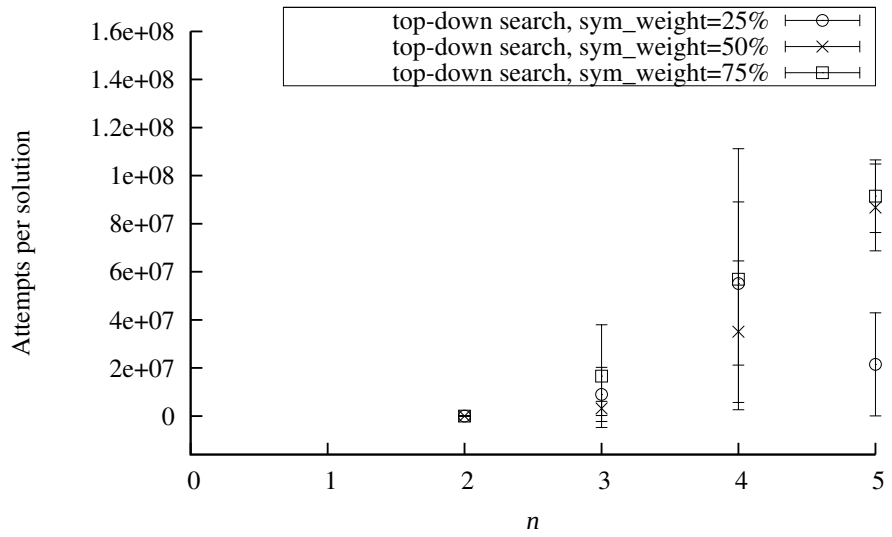



(a) The average number of attempts required before a solution was discovered.

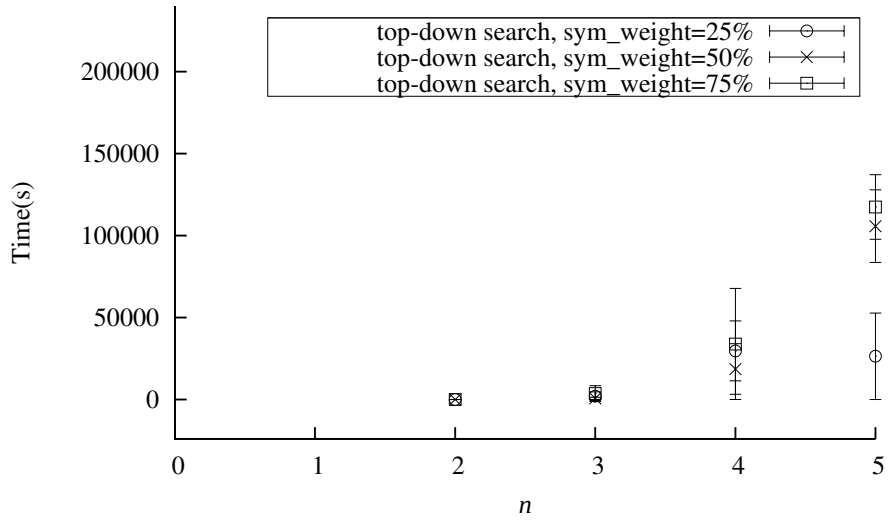



(b) The average CPU time required before a solution was discovered.

**Figure 8.1:** *Top-down n-parity search:* Comparing the performance of *invariance_estimate* and *genetic_search_one* when searching for $n$-parity with $n$ ranging from 2 to 5.

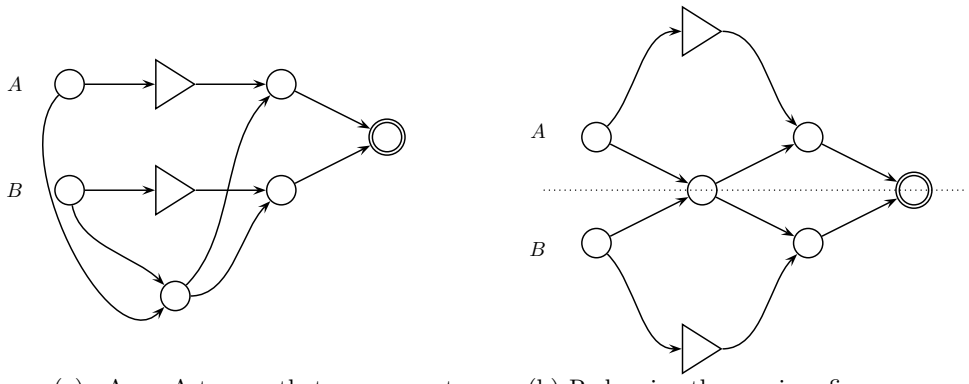



(a) The average number of attempts required before a solution was discovered.




(b) The average CPU time required before a solution was discovered.

**Figure 8.2:** *Top-down n-parity search:* Examining the performance of *invariance_estimate* with three different arguments for *sym_weight*.




(a) An A-type that represents Exclusive-OR.

(b) Redrawing the previous figure to emphasise the solution's mirror symmetry.

**Figure 8.3:** An A-type that represents Exclusive-OR.

a solution to Exclusive-OR, which is a function that is symmetric with respect to $S_2$. We can imagine $S_2$ as a mirror symmetry. In Figure 8.3(b) we redraw $A$ to emphasize that the solution has mirror symmetry. If we reflect $A$ about the dotted line shown then every vertex is mapped to another vertex that is the same (with respect to type (nand machine or delay machine), position (input node or internal node or output node), and incidence (indegree and outdegree)), and every arrow is mapped to another that has the same direction. This notion of a symmetry being imposed via the mapping of elements of a graph is made precise with graph automorphisms (see Section 2.2.1). The above example is appealing because we were able to draw a diagram (Figure 8.3(b)) that had obvious mirror symmetry. However, this may be misleading because a graph may have a symmetry that cannot be illustrated with a diagram.

Consider a function $f : M[k, l] \rightarrow M[p, q]$ that is $B$-invariant for some subgroup $B$ of $S_k$. The idea of the bottom-up scheme is that when we search for an A-type that represents $f$ we only consider A-types that are acted upon by $B$ in such a way that an A-type's input nodes are permuted in the obvious way by $B$. Furthermore, an A-type's input nodes are mapped to themselves, its internal nodes are mapped to themselves, and it's output nodes are mapped to themselves. In our implementation when we search for A-types that represent $f$ we only consider A-types whose graphs are acted upon by subgroups of $B$ of order two. When we devised this approach we initially hoped to only consider A-types whose graphs were acted upon by $B$ in its entirety. We discovered that, in general, this is not possible with A-types. However, we can show that when searching for an A-type that represents $f$ it is always possible to devise an EA that only considers A-types whose graphs are acted upon by a subgroup of $B$, of order two. We believe that such EAs capture some of the symmetry of $f$.

Our bottom-up scheme 'cuts down' the search space by only considering symmetric A-types. We claim that if an A-type has a particular symmetry then it represents a function with that symmetry. We make this precise in the next claim. Let us define an A-type automorphism as an A-type graph automorphism where an A-type's input nodes are mapped to themselves, its internal nodes are mapped to themselves, and its output nodes are maped to themselves.

**Claim 8.2.** Consider an A-type $A$ with an A-type automorphism $\phi$. Let $B$ denote the group that acts on $A$'s input nodes when $\phi$ acts on $A$. The function that $A$ represents is $B$-invariant.

□

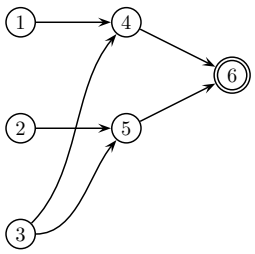
Let us support the above claim by proving that it holds for the following simple example. Consider the A-type, automorphism pair $(A_1, \phi)$ shown in Figure 8.4. According to Claim 8.2 $A_1$ represents a function that is $B$-invariant where $B = (1, (12)) \subseteq S_3$. For this to be true the state of node 6 must be independent of the ordering of input into nodes 1 and 2. Let $q_i(n)$ denote the state of node $n$ at moment $i$. Also let $q_i^\phi(n)$ denote the state of node $n$ at moment $i$ in $\phi(A_1)$. For instance, if at moment $t = 0$ node 1 has state 0 and node 2 has state 1, then $q_0(1) = 0$, $q_0(2) = 1$, $q_0^\phi(1) = 1$, and $q_0^\phi(2) = 0$. With this notation we can say that if Claim 8.2 holds for $A_1$ then $q_i(6) = q_i^\phi(6)$ for all $i \geq \delta = 2$. We now proceed to prove that this is true. Because $A_1$ has a delay of $\delta = 2$ and $A_1$ is feedforward the following holds for all moments $i \geq 2$.

$$
\begin{aligned}
q_i(6) &= q_{i-1}(4) \mathbin{\overline{\wedge}} q_{i-1}(5) \\
&= (q_{i-2}(1) \mathbin{\overline{\wedge}} q_{i-2}(3)) \mathbin{\overline{\wedge}} (q_{i-2}(2) \mathbin{\overline{\wedge}} q_{i-2}(3)) \\
&= (q^\phi_{i-2}(2) \mathbin{\overline{\wedge}} q^\phi_{i-2}(3)) \mathbin{\overline{\wedge}} (q^\phi_{i-2}(1) \mathbin{\overline{\wedge}} q^\phi_{i-2}(3)) \\
&= q^\phi_{i-1}(5) \mathbin{\overline{\wedge}} q^\phi_{i-1}(4) \\
&= q^\phi_{i-1}(4) \mathbin{\overline{\wedge}} q^\phi_{i-1}(5) && \text{since } \overline{\wedge} \text{ is symmetric} \\
&= q_i^\phi(6) && \text{as required.}
\end{aligned}
$$

We hope that this example hints at how one may prove Claim 8.2.

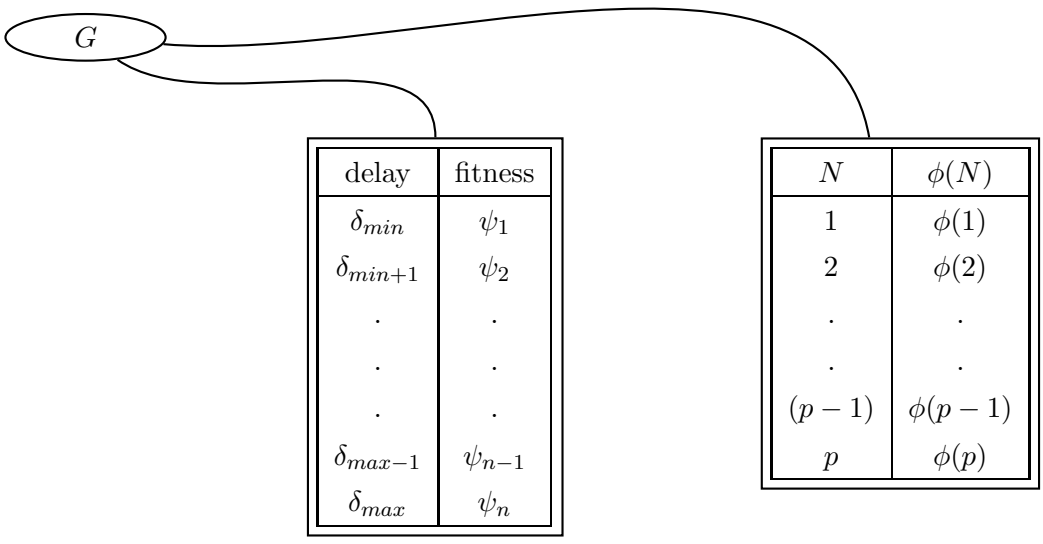
We call our bottom-up search the $\phi$-table algorithm because we implement it by making every A-type object reference a table that specifies an automorphism $\phi$ such that $\phi^2$ is the identity. That is, our A-type objects consist of a graph, a table of fitnesses, and a table of vertex maps—we illustrate this in Figure 8.5.

For example, consider columnwise 3-parity, which is symmetric under $S_3$. Using any of our EAs detailed in Chapter 6 a *candidate* solution to this problem is an A-type that has three input nodes and one output node; for instance, the A-type shown in Figure 8.6(a). Let us consider the subgroup $B' = \{1, (12)\}$ of $S_3$. Consider employing our $\phi$-table search to $(A, \phi)$ pairs where $\phi$ is always an action of $B'$ on $A$'s graph. The pair shown in Figure 8.6(b) has a




| $N$ | $\phi(N)$ |
|---|---|
| 1 | 2 |
| 2 | 1 |
| 3 | 3 |
| 4 | 5 |
| 5 | 4 |
| 6 | 6 |

**Figure 8.4:** The A-type $A_1$, with delay $\delta = 2$, and the automorphism $\phi$ that we use as an example to support Claim 8.2.




| delay | fitness |
|---|---|
| $\delta_{min}$ | $\psi_1$ |
| $\delta_{min+1}$ | $\psi_2$ |
| . | . |
| . | . |
| . | . |
| $\delta_{max-1}$ | $\psi_{n-1}$ |
| $\delta_{max}$ | $\psi_n$ |

| $N$ | $\phi(N)$ |
|---|---|
| 1 | $\phi(1)$ |
| 2 | $\phi(2)$ |
| . | . |
| . | . |
| $(p-1)$ | $\phi(p-1)$ |
| $p$ | $\phi(p)$ |

**Figure 8.5:** An illustration of how each candidate solution is an A-type graph $G$, a list of fitnesses, and an $\phi$-table. Note that since the $\phi$-table has $p$ rows $G$ must have $p$ nodes.

trivial automorphism that fixes all nodes. This pair is not considered in our $\phi$-table algrothim because all of the A-type's input nodes are fixed. The pair shown in Figure 8.6(c) has a non-trivial mapping on the input nodes; consequently, this pair can be a candidate solution for our $\phi$-table algrothim. It is obvious that, for any given delay, all three A-types shown in Figure 8.6 process information identically. However, the $(A, \phi)$ pair in in Figure 8.6(c) also encodes symmetry from $B$. Our $\phi$-table algorithm is an EA that makes use of this property as follows. When searching for an A-type that represents a function that is $B$-invariant for some permutation group $B$ the $\phi$-table algorithm implements the following two rules.

- ⋆ The initial population is composed of $(A, \phi)$ pairs that have a non-trivial symmetry of a subgroup of $B$ of order two. For instance, when searching for A-types that represent a $S_3$-invariant function a $\phi$-table search's initial population may be entirely populated by copies of the $(A, \phi)$ pair shown in Figure 8.6(b).
- ⋆ Every mutation operation is a small change to an $(A, \phi)$ pair such that the result still has a symmetry of a subgroup of $B$ of order two. For instance, when searching for A-types that represent a $S_3$-invariant function the $(A, \phi)$ pair shown in Figure 8.7 is a possible mutant from the $(A, \phi)$ pair shown in Figure 8.6(b). However, in such a search the A-type graph shown in Figure 8.8 is clearly not part of any $(A, \phi)$ pair in the hypothesis space.

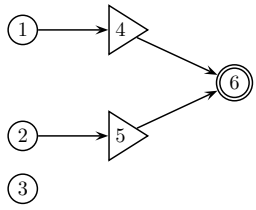
The exclusion of the unsymmetric A-types decreases the hypothesis space. We hypothesize that this provides a mechanism for improving our EAs that search for A-types that represent group invariant functions.

### 8.4.1 A Description of the $\phi$-table Algorithm

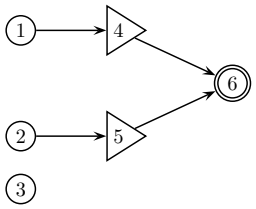
The $\phi$-table algorithm is the same as *mutation_search_one* except that the mutation operators differ. In this section we detail the mutation operator in $\phi$-table. This is a function $(A, \phi) \mapsto (A', \phi')$ which we call *mutate_two*. When devising this operator we made it analogous to *mutate_one* (Section 6.5.2): the A-type $A'$ results from a slight modification of $A$, and $A'$ has a similar number of nodes as $A$; furthermore, the table for $\phi'$ is a slight modification of the table for $\phi$. We construct *mutate_two* by employing five basic operators, namely, *add*, *glue*, *split*, *rewire*, and *delete*. Next we detail each of these five basic operators.

### 8.4.2 Symmetry Operator: *add*

The operator *add* maps an $(A, \phi)$ pair to another $(A', \phi')$ pair such that $A'$ has two more vertices than $A$. That is, if we let $v_A$ denote the vertex set of the graph of $A$ and let $V_{A'}$ denote the vertex set of the graph of $A'$, then $V_{A'} = V_A \bigcup \{v_1, v_2\}$. Furthermore, $\phi'$ is such that $\phi'(v_1) = v_2$, $\phi'(v_2) = v_1$, and if we restrict $\phi'$ to $V_A$ then $\phi' = \phi$. The graph of $A'$ is chosen so that it is a valid A-type graph and the above conditions for $\phi'$ are satisfied. In

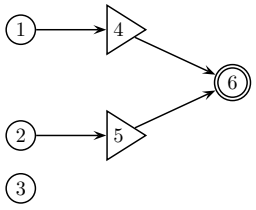



(a) The A-type graph of a possible candidate solution when *genetic_search_one* searches for an A-type that represents 3-parity.




| $N$ | $\phi(N)$ |
|---|---|
| 1 | 1 |
| 2 | 2 |
| 3 | 3 |
| 4 | 4 |
| 5 | 5 |
| 6 | 6 |

(b) The A-type graph $A$ (shown in the above diagram) and an automorphism for the group $(1, (12)) \supseteq S_3$ acting on $A$. Under this automorphism all of $A$'s input nodes are fixed; consequently this pair cannot be a candidate solution in $\phi$-table.




| $N$ | $\phi(N)$ |
|---|---|
| 1 | 2 |
| 2 | 1 |
| 3 | 3 |
| 4 | 5 |
| 5 | 4 |
| 6 | 6 |

(c) The A-type graph $A$ and another automorphism. This pair can be a candidate solution when *genetic_search_one* searches for an A-type that represents 3-parity.

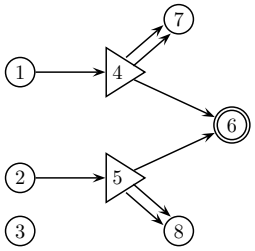
**Figure 8.6:** Illustrating a possible candidate solutions when we use our $\phi$-table algorithm to search for an A-type that represents 3-parity.

Figure 8.9 we illustrate how *add* operates on a simple $(A, \phi)$ pair. Table 8.3 gives a description of *add*.

### 8.4.3 Symmetry Operator: *glue*

The operator *glue* maps an $(A, \phi)$ pair to another $(A', \phi')$ pair such that $A'$ has one fewer vertices than $A$. The pair $(A', \phi')$ is constructed by copying $A$, choosing two nodes $n$ and $\phi(n)$, and replacing these with a new node that is fixed under $\phi'$. The resulting graph is chosen to be a valid A-type graph that is only a 'slight' modification of $A$. In Figure 8.10 we illustrate how *glue* operates on a simple $(A, \phi)$ pair.

Before we describe the *glue* operator let us introduce some notation. When we glue two




| $N$ | $\phi(N)$ |
|---|---|
| 1 | 2 |
| 2 | 1 |
| 3 | 3 |
| 4 | 5 |
| 5 | 4 |
| 6 | 6 |
| 7 | 8 |
| 8 | 7 |

**Figure 8.7:** A mutation from the $(A, \phi)$ pair shown in Figure 8.6(b).

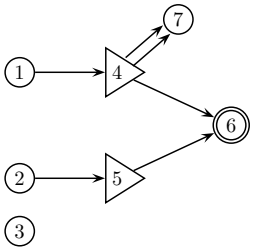
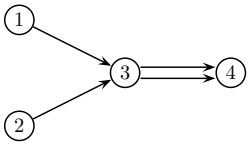
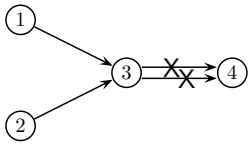



**Figure 8.8:** An $A$ type graph that does not belong to any candidate solution when we use $\phi$-table to search for A-types that represent 3-parity.

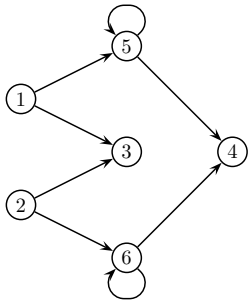



| $N$ | $\phi(N)$ |
|---|---|
| 1 | 2 |
| 2 | 1 |
| 3 | 3 |
| 4 | 4 |

(a) The original (A-type, automorphism) pair.




(b) Manipulating the original A-type.




| $N$ | $\phi(N)$ |
|---|---|
| 1 | 2 |
| 2 | 1 |
| 3 | 3 |
| 5 | 6 |
| 6 | 5 |
| 4 | 4 |

(c) The mutant (A-type, automorphism) pair.

**Figure 8.9:** A simple example of *add*: adding two nodes which we label 5 and 6. Note that both the A-type graph and the $\phi$-table are altered.

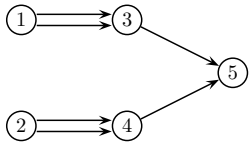
**add**$(P)$ is an operator that adds two nodes to the A-type $A$ in a specified (A-type, automorphism) pair $P = (A, \phi)$. In doing so arrows are removed from $A$ and new arrows introduced such that the newly added nodes have no vacancies and $A$'s symmetry is preserved.

| Parameters | | |
|---|---|---|
| *Type* | *Parameter* | *Description* |
| (A-type, automorphism) | $(A, \phi)$ | Object containing the A-type to which we add two nodes. |

The algorithm has the following steps

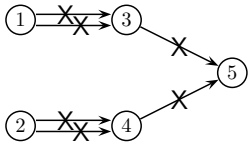
1. *Adding nodes:* Add two nodes $n1$, $n2$ to $A$. Both $n1$ and $n2$ are nand machines or both $n1$ and $n2$ are delay machines. Also add $\phi(n1) = \phi(n2)$ to $P$'s $\phi$-table.
2. *Ensuring that the added nodes have a non-zero outdegree:*
   a) Chose an arrow $(s, t)$ from $A$.
   b) Remove the two arrows $(s, t)$ and $(\phi(s), \phi(t))$ from $A$. If $t$ is fixed and $t$ is a delay machine then this step fails because $(s, t) = (\phi(s), \phi(t))$. In this case another arrow is randomly selected for $(s, t)$. If $A$ does not contain any arrows that let this step succeeds then *add*$(P)$ fails.
3. *Ensuring that the added nodes have the correct indegree:*
   a) Choose a non-input node $s$ from $A$.
   b) Add the arrows $(s, n1)$ and $(\phi(s), n2)$ to $A$.
   c) If $n1$ and $n2$ are nand machines then we repeat steps 3.a) and 3.b) once.
4. Return $P$.

**Table 8.3:** A description of *add*.

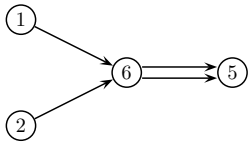



| $N$ | $\phi(N)$ |
|---|---|
| 1 | 2 |
| 2 | 1 |
| 3 | 4 |
| 4 | 3 |
| 5 | 5 |

(a) The original (A-type, automorphism) pair.




(b) Manipulating the original A-type.




| $N$ | $\phi(N)$ |
|---|---|
| 1 | 2 |
| 2 | 1 |
| 6 | 6 |
| 5 | 5 |

(c) The mutant (A-type, automorphism) pair.

**Figure 8.10:** A simple example of *glue*: gluing the nodes 3 and 4. Note that both the A-type graph and the $\phi$-table are altered.

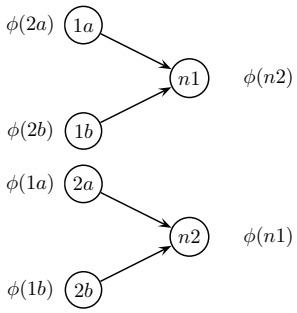



**Figure 8.11:** *Notation:* A subgraph of an A-type in which we glue the nodes $n1$ and $n2$. This illustrates our convention for labelling the sources of arrows that enter the nodes $n1$ and $n2$.

nodes $n1$, $n2$ together it is necessary to keep track of the sources of arrows entering $n1$ and $n2$. If $n1$ and $n2$ are delay machines then let $(1a, n1)$ denote the arrow entering $n1$, and let $(2a, n2)$ denote the arrow entering $n2$. If $n1$ and $n2$ are nand machines then let $(1a, n1)$ and $(1b, n1)$ denote the arrows entering $n1$, and let $(2a, n2)$, $(2b, n2)$ denote the arrow entering $n2$. Where $\phi(1a) = 2a$ and $\phi(1b) = 2b$. We illustrate this convention in Figure 8.11. Table 8.4 gives a description of *glue*.

Note that step 1.d) in *glue* is rather involved: For a given pair $P$ with nodes $n1$ and $n2$ often there are several possible return pairs. For example, in Figure 8.12 illustrates three possible outputs from a given glue operation.

**glue**$(P, n1, n2)$ is an operator that modifies nodes in the specified (A-type, automorphism) pair $P = (A, \phi)$. This modification is the replacement of the nodes $n1$ and $n2$ by a new node such that $A$'s symmetry is preserved. If this is not possible then the A-type is returned unaltered.

Parameters

| *Type* | *Parameter* | *Description* |
|---|---|---|
| (A-type, automorphism) | $(A, \phi)$ | Object whose A-type has nodes to be glued. |
| Node | $n1$ | Internal node of $A$ where $n1 = \phi(n2)$. |
| Node | $n2$ | Internal node of $A$ where $n2 = \phi(n1)$. |

The algorithm has the following steps

1. *Exit condition:* If $n1$ and $n2$ are delay machines and if $1a \neq 2a$ then the glue is impossible.

   otherwise

   a) *Adding node:* Construct a node $n$ that is the same type (delay machine or nand machine) as $n1$ and $n2$. Add $n$ to $A$. Add $\phi(n) = n$ to $P$'s $\phi$-table.

   b) *Removing exit arrows:* Remove each arrow $(n1, ti)$ exiting $n1$ and add the arrow $(n, ti)$. Similarly, remove each arrow $(n2, ui)$ exiting $n2$ and add the arrow $(n2, ui)$.

   c) Remove $n1$ and $n2$, and remove all arrows entering these nodes. Also remove $\phi(n1) = n2$ from $P$'s $\phi$-table.

   d) *Ensuring that n has the correct indegree:*
   Repeat until $n$'s indegree is one if $n$ is a delay machine and two if $n$ is a nand machine.

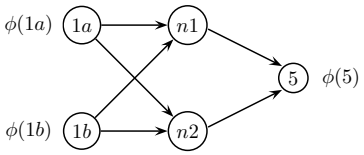
      i. choose a node $s$ from $\{1a, 1b, 2a, 2b\}$.

      ii. If $s$ and $\phi(s)$ are distinct and $n$ currently has one vacancy (an indegree one less than its correct indegree) then go back to step d)i.

      iii. Add the arrows $(s, n)$ and $(\phi(s), n)$. Note that $(s, n)$ may equal $(\phi(s), n)$.

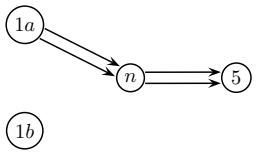
2. Return $P$

**Table 8.4:** A description of *glue*.



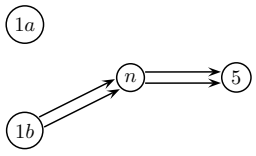

| $N$ | $\phi(N)$ |
|---|---|
| 1a | 1a |
| 1b | 1b |
| n1 | n2 |
| n2 | n1 |
| 5 | 5 |

(a) The original A-type. Note that all nodes except $n1$ and $n2$ are fixed.



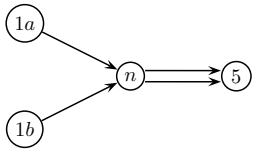

| $N$ | $\phi(N)$ |
|---|---|
| 1a | 1a |
| 1b | 1b |
| n | n |
| 5 | 5 |

(b) A possible output from *glue*.




| $N$ | $\phi(N)$ |
|---|---|
| 1a | 1a |
| 1b | 1b |
| n | n |
| 5 | 5 |

(c) A possible output from *glue*.




| $N$ | $\phi(N)$ |
|---|---|
| 1a | 1a |
| 1b | 1b |
| n | n |
| 5 | 5 |

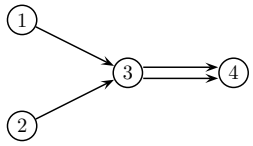
(d) A possible output from *glue*.

**Figure 8.12:** A simple example illustrating that several possible outputs may result from a single *glue* call.

### 8.4.4 Symmetry Operator: *split*

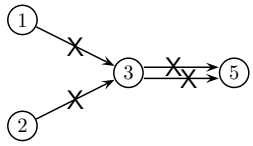
The operator *split* is the 'opposite' of *glue*. We designed *split* so that, given a $(A, \phi)$ pair, we can apply *glue* and *split* in sequence and the result is identical to $(A, \phi)$. Note that, in general, this outcome is not guaranteed because applying *glue* to a given $(A, \phi)$ pair does not return a unique result. This is also the case for *split*. In Figure 8.13 we illustrate how *split* operates on a simple $(A, \phi)$ pair. Table 8.5 gives a description of *split*. Note that in this description we re-employ the notation illustrated in Figure 8.11.

Note that step b)i. in our description of *split* is rather involved. Figure 8.14 illustrates an application of the rules in this step.

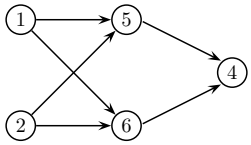



| $N$ | $\phi(N)$ |
|---|---|
| 1 | 2 |
| 2 | 1 |
| 3 | 3 |
| 4 | 4 |

(a) The original (A-type, automorphism) pair.




(b) Manipulating the original A-type.




| $N$ | $\phi(N)$ |
|---|---|
| 1 | 2 |
| 2 | 1 |
| 5 | 6 |
| 6 | 5 |
| 4 | 4 |

(c) The mutant (A-type, automorphism) pair.

**Figure 8.13:** A simple example of *split*: splitting node 3. Note that both the A-type graph and the $\phi$-table are altered.

### 8.4.5 Symmetry Operator: *rewire*

The operator *rewire* maps an $(A, \phi)$ pair to another $(A', \phi')$ pair. The graph of $A$ is a slight modification of the graph of $A'$. These two graphs have the same number of vertices. The graphs differ by a slight modification that is analogous to *mutate_constant* (see Section 6.5.2): we select an arrow and 'move one end' so that it has a different source node. More precisely, we remove an arrow $(s, t)$ and we insert an arrow $(s', t)$, where $s \neq s'$. When we perform this task we require $A'$ to have the same symmetry as $A$. This often involves the alteration of two arrows $(s, t)$ and $(\phi(s), \phi(t))$.

In Figure 8.15 we illustrate *rewire* operating on a simple $(A, \phi)$ pair. Table 8.6 gives a description of *rewire*. Note that we employ the subroutine *change_source* (see Table 8.7), which is also employed in our description of the next symmetry operator: *delete*.

**split**$(P, n)$ is an operator that modifies the A-type in a specified (A-type, automorphism) pair. This modification replaces a fixed internal node with two nodes that are mapped to one another under $\phi$.

| Parameters | | |
|---|---|---|
| *Type* | *Parameter* | *Description* |
| (A-type, automorphism) | $(A, \phi)$ | Object containing the A-type to which we add two nodes. |
| Node | $n$ | Fixed internal node of $A$. |

The algorithm has the following steps

1. *Exit condition:* If any targets of $n$ are fixed delay machines then $n$ cannot be split and P will be returned unaltered.

   otherwise

   a) *Adding two nodes:* Construct two nodes $n1$ and $n2$ that have the same type (delay machine or nand machine) as $n$. Add $n1$ and $n2$ to $A$. Add $\phi(n1) = n2$ to $P$'s $\phi$-table.

   b) *Rewiring arrows from $n$ to $n1$ and $n2$:*
   Consider arrows that $(n, ti)$ that exit $n$; iterate through each target $ti$.

      i. If $ti$ is a nand machine and if $ti = \phi(ti)$ then delete both arrows that enter $ti$, and add the two arrows $(n1, ti)$ and $(n2, ti)$.

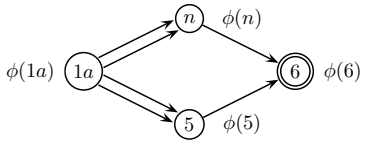
      ii. If $ti$ is not fixed then delete $(n, ti)$ and $(n.\phi(ti))$. Furthermore, choose one of the following two cases: add $(n1, ti)$ and $(n2, \phi(ti))$, or add $(n1, \phi(ti))$ and $(n2, ti)$.

   c) *Ensuring that $n1$ and $n2$ have the correct indegree:*
   If $n$ is a delay machine then remove the arrow $(1a, n)$ that enters $n$; furthermore add the arrows $(1a, n1)$ and $(1a, n2)$. If $n$ is a nand machine then delete the arrows $(1a, n)$ and $(1b, n)$ that enter $n$. Furthermore, add the arrows $(1a, n1)$, $(1b, n1)$, $(1a, n2)$ and $(1b, n2)$.

   d) *Deleting $n$:* Delete all arrows that enter $n$, delete all arrows that exit $n$, and delete $n$.

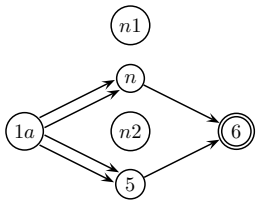
2. Return $P$

**Table 8.5:** A description of *split*.



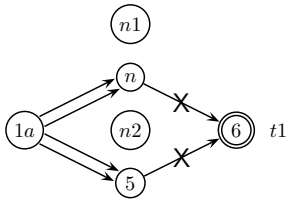

| $N$ | $\phi(N)$ |
|---|---|
| 1a | 1a |
| n | n |
| 5 | 5 |
| 6 | 6 |

(a) The original $(A, \phi)$ object with all nodes in the $A$ fixed under $\phi$.



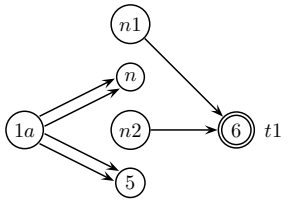

| $N$ | $\phi(N)$ |
|---|---|
| 1a | 1a |
| n1 | n2 |
| n | n |
| n2 | n1 |
| 5 | 5 |
| 6 | 6 |

(b) Adding two nodes $n1$ and $n2$, with $n1 = \phi(n2)$.



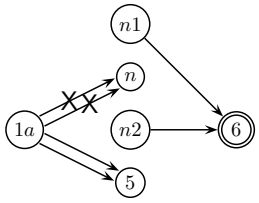

| $N$ | $\phi(N)$ |
|---|---|
| 1a | 1a |
| n1 | n2 |
| n | n |
| n2 | n1 |
| 5 | 5 |
| 6 | 6 |

(c) Let $(n, t1)$ denote the arrow that exits $n$. We delete all arrows that enter $t1$.



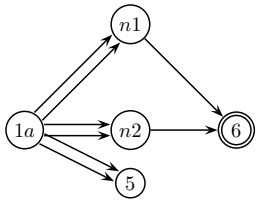

| $N$ | $\phi(N)$ |
|---|---|
| 1a | 1a |
| n1 | n2 |
| n | n |
| n2 | n1 |
| 5 | 5 |
| 6 | 6 |

(d) Adding the arrows $(n1, t1)$ and $(n2, t1)$.




| $N$ | $\phi(N)$ |
|---|---|
| 1a | 1a |
| n1 | n2 |
| n | n |
| n2 | n1 |
| 5 | 5 |
| 6 | 6 |

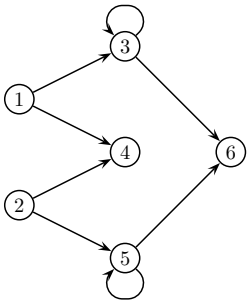
(e) Deleting $n$ and the arrows that enter $n$.




| $N$ | $\phi(N)$ |
|---|---|
| 1a | 1a |
| n1 | n2 |
| n2 | n1 |
| 5 | 5 |
| 6 | 6 |

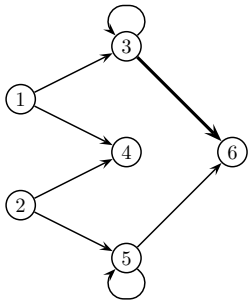
(f) The return $(A, \phi)$ object.

**Figure 8.14:** Illustrating step b)i. in our description of *split*. Applying *split* on a particular pair $(A, \phi)$.



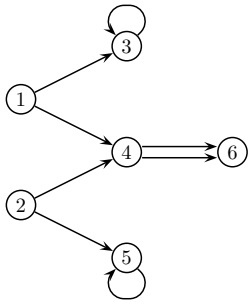

| $N$ | $\phi(N)$ |
|---|---|
| 1 | 2 |
| 2 | 1 |
| 3 | 5 |
| 5 | 4 |
| 6 | 3 |
| 4 | 6 |

(a) The original (A-type, automorphism) pair.




| $N$ | $\phi(N)$ |
|---|---|
| 1 | 2 |
| 2 | 1 |
| 3 | 5 |
| 4 | 4 |
| 5 | 3 |
| 6 | 6 |

(b) Manipulating the original A-type.




| $N$ | $\phi(N)$ |
|---|---|
| 1 | 2 |
| 2 | 1 |
| 3 | 5 |
| 4 | 4 |
| 5 | 3 |
| 6 | 6 |

(c) The mutant (A-type, automorphism) pair.

**Figure 8.15:** A simple example of *rewire*: choosing the arrow $(3, 6)$, choosing a new source for this arrow; consequently, changing the source of the arrow $(5, 6)$. Note that both the A-type graph and the $\phi$-table are altered.

**rewire**$(P)$ is an operator that modifies the specified (A-type, automorphism) pair $P$. This modification is the removal of an arrow in $P$ and the insertion of a new arrow such that $P$'s symmetry is preserved. If this is not possible then the $P$ is returned unaltered. Note that this operator calls the *change_source* subroutine, whose description follows the present description.

| Parameters | | |
|---|---|---|
| *Type* | *Parameter* | *Description* |
| (A-type, automorphism) | $(A, \phi)$ | Object containing the A-type to which we add two nodes. |

The algorithm has the following steps

1. *Iterating through all arrows:* Construct a set $R = \{A'sarrows\}$
   Repeat
   a) Randomly select and remove an arrow $(s, t)$ from $R$.
   b) try $P \leftarrow change\_source(P, s, t)$
   if this call succeeds then $rewire(P)$ exits successfully.
   otherwise
   if $R$ is empty then $rewire(P)$ fails.
2. Return $P$

**Table 8.6:** A description of *rewire*.

### 8.4.6 Symmetry Operator: *delete*

Finally, the operator *delete* accepts a pair $(A, \phi)$ and returns a pair $(A', \phi')$ such that $A'$ has one or two fewer nodes than $A$. This is achieved by the deletion of a node $n$ from a copy of $A$. This operation ensures that $A'$ is a valid A-type and has the same symmetry as $A$. In general two nodes $n$ and $\phi(n)$ are removed; furthermore, an appropriate rewiring is performed. In Figure 8.16 we illustrate how *delete* operates on a simple $(A, \phi)$ pair. Table 8.8 provides a description of *delete*.

### 8.4.7 An Outline of *mutate_two*

Our mutation operator for $\phi$-table, which we call *mutate_two*, randomly selects one of the five symmetry operators described above. The only restriction on this process is that in some cases the input's A-type has too few internal nodes for either *glue* or *delete* to be applied. If

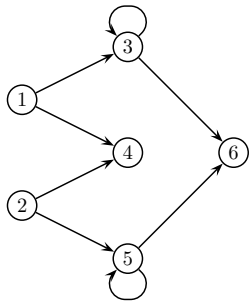
**change_source**$(P, s, t)$ is an operator that modifies the A-type $A$ which is in the specified pair $P = (A, \phi)$. This modification is the removal of the arrow $(s, t)$ from $A$ and the insertion of a different arrow $(r, t)$ such that $r$ is randomly selected and $A$'s symmetry is preserved. If this is not possible then $P$ is returned unaltered.

| Parameters | | |
|---|---|---|
| *Type* | *Parameter* | *Description* |
| (A-type, automorphism) | $(A, \phi)$ | Object containing the A-type to which we add two nodes. |
| Node | $s$ | Source of the arrow that is to be removed. |
| Node | $t$ | Target of the arrow that is to be removed. |

The algorithm has the following steps

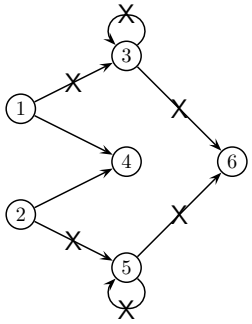
1. If $t$ is fixed and $t$ is a nand machine.
   a) Remove the two arrows that enter $n$.
   b) *Possible new sources:* Construct a set $S$ of possible new sources.
   If $t$ is an output node then $S = \{A$'s internal nodes $\} - s - \phi(s)$
   otherwise, $t$ is an internal node, then $S = \{A$'s internal nodes $\} \cup \{A$'s internal nodes $\} - s - \phi(s)$
   If $S$ is the empty set then this algorithm fails.
   c) *Adding new arrows:* $r \rightarrow$ randomly chosen node from $S$.
   Add the two arrows $(r, t)$ and $(\phi(r), t)$. Note that if $r = \phi(r)$ then these arrows are still distinct.
2. If $t$ is fixed and $t$ is a delay machine.
   a) Remove the arrow $(s, t)$.
   b) *Possible new sources:* Construct a set $S$ of possible new sources.
   If $t$ is an output node then $S = \{A$'s fixed internal nodes$\} - s - \phi(s)$
   otherwise, $t$ is an internal node, then $S = \{A$'s fixed internal nodes $\} \cup \{A$'s fixed internal nodes $\} - s - \phi(s)$
   If $S$ is the empty set then this algorithm fails.
   c) *Adding new arrows:* $r \rightarrow$ randomly chosen node from $S$.
   Add the arrow $(r, t)$.

**Table 8.7:** A subroutine for *rewire* (also for *delete*) that 'moves the tail' end of a specified arrow.



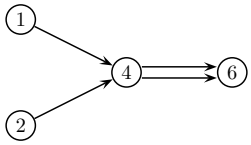

| $N$ | $\phi(N)$ |
|---|---|
| 1 | 2 |
| 2 | 1 |
| 3 | 5 |
| 5 | 4 |
| 6 | 3 |
| 4 | 6 |

(a) The original (A-type, automorphism) pair.




| $N$ | $\phi(N)$ |
|---|---|
| 1 | 2 |
| 2 | 1 |
| 3 | 3 |
| 5 | 6 |
| 6 | 5 |
| 4 | 4 |

(b) Manipulating the original A-type.




| $N$ | $\phi(N)$ |
|---|---|
| 1 | 2 |
| 2 | 1 |
| 4 | 4 |
| 6 | 6 |

(c) The mutant (A-type, automorphism) pair.

**Figure 8.16:** A simple example of *delete*: deleting node 3; consequently, deleting node $\phi(3) = 5$. Note that both the A-type graph and the $\phi$-table are altered.

3. If $t$ is free.

   Remove the two arrows $(s, t)$ and $(\phi(s), \phi(t))$.

   a) *Possible new sources:* Construct a set $S$ of possible new sources.
   If $t$ is an output node then $S = \{A$'s internal nodes $\} - s - \phi(s)$
   otherwise, $t$ is an internal node, then $S = \{A$'s internal nodes $\} \cup \{A$'s internal nodes $\} - s - \phi(s)$
   If $S$ is the empty set then this algorithm fails.

   b) *Adding new arrows:* $r \rightarrow$ randomly chosen node from $S$.
   Add the two arrows $(r, t)$ and $(\phi(r), \phi(t))$.

4. Return $P$

**Table 8.7:** A subroutine for *rewire* (also for *delete*) that 'moves the tail end of a specified arrow' (cont.).

we do not avoid such cases then *mutate_two* may remove all internal nodes giving an invalid A-type. Table 8.9 gives a description of *mutate_two*.

### 8.4.8 Testing the $\phi$-table Algorithm

We examine the performance of our bottom-up approach when it searches for clamped A-types that represent $n$-parity—as we did with the top-down approach. We compare $\phi$-table to *genetic_search_one* when both algorithms were employed to search for $n$-parity with $n \in \{2, 3, 4, 5\}$. The results for our searches with *genetic_search_one* were presented in Section 7.4.6. In that section we specified the arguments chosen for the parameters of our searches. These arguments were also used when we tested the $\phi$-table EA, with two exceptions. First, when searching with $\phi$-table we only performed 10 trials for each $n$. Because this smaller number of trials gave conclusive results and because the trials took a considerable time to complete, we believe that our chosen number of trials is reasonable. Second, the initial population of all $\phi$-table searches consisted of 100 copies of the $(A, \phi)$ pair shown in Figure 8.17. This ensures that every member of the initial population has the symmetry of $((12), 1) \subseteq S_n$ for all of our $n$-parity searches. The results are shown in Figure 8.1. For all trials where $\phi$-table searched for 5-parity it failed to find a solution before termination (after $10^8$ attempts). These results show that our $\phi$-table search significantly underperforms in comparison to *genetic_search_one*. Considering the substantial effort required for its implementation, $\phi$-table performs very poorly. In spite of this result, we believe that it is worthwhile to continue researching $\phi$-table, because other researchers have achieved positive results by imposing symmetries directly on ANNS [14] [85] and A-types offer an excellent

**delete**$(P, n)$ is an operator that attempts to modify nodes in the A-type $A$ which is in the specified (A-type, automorphism) pair $P = (A, \phi)$. This modification is the deletion of the node $n$ such that $A$'s symmetry is preserved; if this is not possible then the A-type is returned unaltered.

| Parameters | | |
|---|---|---|
| *Type* | *Parameter* | *Description* |
| (A-type, automorphism) | $(A, \phi)$ | Object containing the A-type from which we delete a node. |
| Node | $n$ | Element of $A$'s internal nodes to be deleted. |

The algorithm has the following steps

1. *Determining arrows to be re-sourced:* Construct the following set.
   $S = \{$ arrows that exit $n\}$. If there are two distinct arrows $(n, ti)$,$(n, ti)$' then only one such arrow is chosen (This caters for step 1.a) in *change_source*).

2. *Re-sourcing arrows:* Iterate through all arrows $(n, ti)$ in $S$.
   try *change_source*$(P, n, ti)$
   if this fails then *delete*$(P, n)$ fails.

3. *Deleting n:* Delete all arrows that exit $n$, delete all arrows that enter $n$, and delete $n$. Similarly, delete all arrows that exit $\phi(n)$, delete all arrows that enter $\phi(n)$, and delete $\phi(n)$. Delete $\phi(\phi(n)) = n$ from $p$'s $\phi$-table.

**Table 8.8:** A description of *delete*. Note step 3 in this algorithm—where we rewire arrows that exit $N$ or $f(N)$. If we have two distinct source nodes and a target node that has a sole vacancy then there exists freedom in our rewiring; we make a random selection in such situations.

**mutate_two**$((A, \phi))$ is a mutation operator that, given a $(A, \phi)$ pair, returns a slightly modified $(A', \phi')$ pair such that $A$ and $A'$ have the same symmetry.

| Parameters | | |
|---|---|---|
| *Type* | *Parameter* | *Description* |
| (A-type, automorphism) | $(A, \phi)$ | The original (A-type, automorphism) pair. |

The algorithm has the following steps

1. *Choosing a symmetry operator:*

   N $\leftarrow$ the number of internal nodes in $A$.

   if(N $= 1$) then

   R $\leftarrow$ element randomly selected from $\{0, 1, 2\}$

   otherwise

   R $\leftarrow$ element randomly selected from $\{0, 1, 2, 3, 4\}$

   a) *Performing the selected symmetry operator*

   if(R $= 0$) then
   $(A', \phi') \leftarrow add((A', \phi'))$
   if(R $= 1$) then
   $(A', \phi') \leftarrow split((A', \phi'))$
   if(R $= 2$) then
   $(A', \phi') \leftarrow rewire((A', \phi'))$
   if(R $= 3$) then
   $(A', \phi') \leftarrow delete((A', \phi'))$
   if(R $= 4$) then
   $(A', \phi') \leftarrow glue((A', \phi'))$

2. Return $(A', \phi')$.

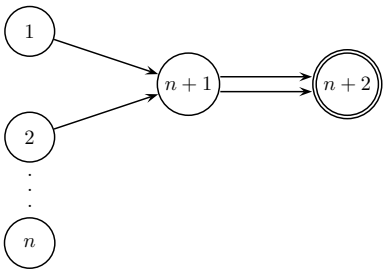
**Table 8.9:** A description of *mutate_two*

test-bed for these ideas.

## 8.5 Conclusions

In this chapter we achieved the following.

⋆ We implemented an EA, which we called *invariance_estimate*, that estimated each



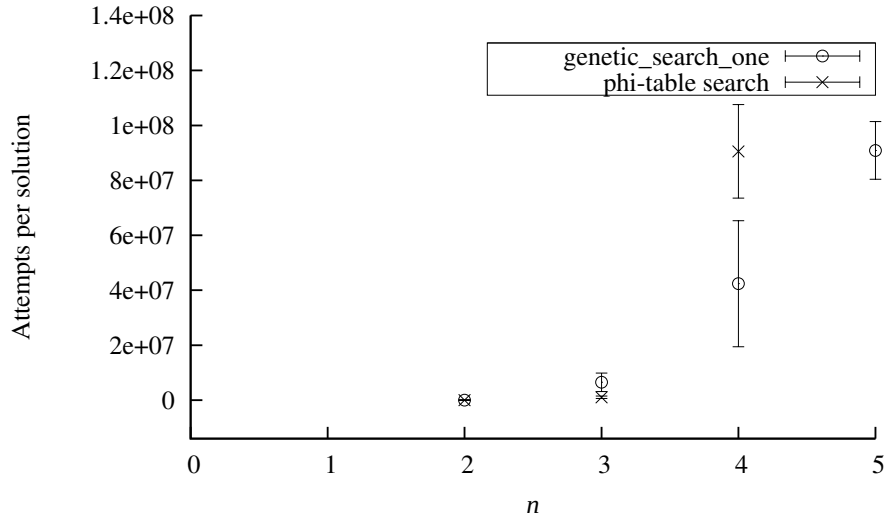

| $N$ | $\phi(N)$ |
|---|---|
| 1 | 2 |
| 2 | 1 |
| . | . |
| . | . |
| . | . |
| $n$ | $n$ |
| $n+1$ | $n+2$ |
| $n+2$ | $n+2$ |

**Figure 8.17:** When we employ $\phi$-table to search for clamped A-types that represent $n$-parity all $(A, \phi)$ pairs in the initial population are like the pair illustrated. There is only one internal node and the arrows entering that node are the only arrows not fixed under $\phi$. This automorphism gives an action of the group $(1, (12)) \subseteq S_n$. on $A$.

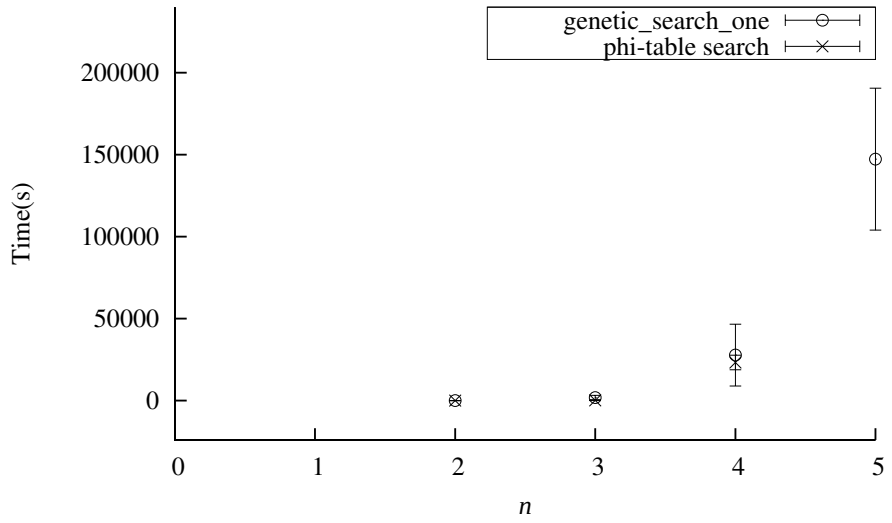
A-type candidate solution's group invariance and made this estimate contribute to the candidate solution's fitness. However, we require further investigation to verify whether *invariance_estimate* can be optimized to outperform *genetic_search_one*.

⋆ We implemented an EA that evolved (A-type, automorphism) pairs, which we called the $\phi$-table algorithm. Every candidate solution in this search is an A-type with a permutation symmetry (of order two). Currently, our $\phi$-table significantly underperforms in comparison to *genetic_search_one* when searching for A-types that represent $n$-parity.

⋆ We have demonstrated that A-types can be used to implement, and test, sophisticated ideas in machine learning.




(a) The average number of attempts required before a solution was discovered.




(b) The average CPU time required before a solution was discovered.

**Figure 8.18:** *Bottom-up n-parity search:* Comparing the performance of the $\phi$-table search and *genetic_search_one* when searching for $n$-parity with $n$ ranging from 2 to 5. Note that all trials where $\phi$-table searched for 5-parity it failed to find a solution before termination (after $10^8$ attempts).

# 9 Conclusions

## 9.1 Aims of this Chapter

In this chapter we summarize this project. Also, we make suggestions for future research; in particular, we speculate about evolving evolutionary operators for A-types.

## 9.2 Conclusions

In this project we achieved the following results.

- ⋆ We took the historically important idea of Turing's A-type ANNs and implemented it to test some novel schemes. These basic networks can operate in a sequential dynamic fashion yet relatively few researchers make use of them. We interpreted Turing's networks to consist of nand machines and delay machines. We employed the language of finite state machines and graph theory to precisely specify our interpretation of Turing's A-types. We demonstrated that the inclusion of delay machines allow A-types to operate in a sequential fashion.

- ⋆ We devised a computer program that lets the user conduct EAs with hypothesis spaces of discrete ANNs. This program was implemented in Java and it allows the user to employ one of several EAs with one of several types of ANN.

- ⋆ We devised and implemented an EA with a hypothesis space of A-types. We tested this algorithm on simple benchmark problems. We showed that our EA out performs a blind search. This is evidence that evolving a population of A-types is a useful approach.

- ⋆ In our EA we included an intricate crossover scheme that uses some simple ideas from graph theory; such as, connectedness, subgraphs, and boundaries. We did this to construct a crossover operator that performs better than a macromutation operator. With two out of four simple benchmark tests we showed that our crossover is a beneficial operator for our EA. Furthermore, we repeated these two successful tests with a crossover operator that crosses one parent with a random parent (a headless chicken search). Our original EA outperformed the headless chicken EA. This showed that, for some problems, our crossover is more beneficial than a macromutation operator.

- ⋆ We hypothesized that ideas from group theory could be used to improve evolutionary searches for A-type solutions to a particular class of problems (problems that are invariant under $S_2$ permutations of the input variables). To test this hypothesis we devised and implemented two EAs. The first algorithm estimated every candidate solution's symmetry and used this estimate as part of the fitness function. Less 'symmetric' candidate solutions tended to be less fit. The second algorithm evolved a population of A-types that had the desired symmetry 'built-in'. We tested both algorithms on the $n$-parity problem and compared the results with our previous EA's performance on $n$-parity. We failed to find evidence to support our hypothesis.
- ⋆ With both our crossover investigations and our group theoretic investigations we demonstrated that A-types are useful test-beds for investigating ideas about evolving ANNs. The simplicity of A-types' neurons allow researchers to implement intricate ideas.

## 9.3 Future Research

In this section we suggest directions for future research.

### 9.3.1 Our Group Theoretical Ideas

The author believes that our ideas of evolving (A-type, graph automorphism) pairs are worth further investigation; in spite of the disappointing results that we currently have.

### 9.3.2 Evolving Evolutionary Operators

Here we propose that it is worthwhile to investigate the evolution of evolutionary operators for A-types.

The algorithm *genetic_search_one* has many parameters that require arguments; for instance, the population size, the selection rules, and the mutation to crossover ratio. When we tested this algorithm (see Chapter 7) we were rather unsystematic when we assigned these parameters. We performed ad hoc tests to determine what seemed to be reasonable values. Finding optimal parameters is a computationally expensive task. Furthermore, these parameters may be particular to each benchmark task. There are two reasons why we were unsystematic when we assigned parameters for our search. First, we believe that it was pragmatic to start our investigations with what our ad hoc tests suggested were reasonable variables. Second, we believe that the discovery of optimal parameters is a special case of methods that we had hoped to include in this project, namely the evolution of an evolutionary search for A-types.

The evolution of parameters of a search is an established technique in evolutionary computing [47, ch 4]. Many researchers have extended this idea to include the evolution of evolutionary operators [86]. We can motivate this by considering biological analogies. For

instance, biological crossover is homologous (that is, only certain parts of a genotype can be exchanged) and this is the reason that biological crossover is beneficial [43, p50]. The mechanics of biological recombination is truly awe inspiring and it too is a consequence of evolution. Evolutionary search for evolutionary operators is an established field of research. In terms of evolving networks, Teller's research [51] [87] is of particular interest. Teller solved signal classification tasks by evolving two populations simultaneously. One population was a set of programs; these programs—unlike Koza's tree chromosomes—were represented with graphs. The other population was a set of evolutionary operators that operate on the programs.

The decision to evolve evolutionary operators is well motivated, but this logic may lead to a recursive procedure: each population of operators may require yet another population of operators. Again, one can turn to biology for inspiration. The staggering complexity of biological life is ultimately a consequence of molecular chemistry. This motivates the field of artificial chemistry [88]. This aims to discover algorithms by prescribing a set of basic rules and evolving this set until a desired algorithm emerges. In his doctoral thesis Teuscher [89, sec 8.2] articulates the potential usefulness of artificial chemistries to program artificial neural networks. However, he states that discovering an algorithm for implementing artificial chemistries is a non-trivial task. Furthermore, he states that the detailed calculations required are likely to be computationally intractable on current computers.

Applying artificial chemistries to A-types has great appeal. However, we decided that an easier task was to try to co-evolve evolutionary operators in a manner analogous to Teller's research. Accepting this as a challenge—especially in light of the time constraints of a doctoral program—we began an introductory investigation. This research is incomplete but the author believes that it is valid to speculate about the evolution of evolutionary operators for A-types. We do this below.

- ⋆ Our results in Chapter 7 show that, for some problems, our crossover operator is more useful than a macromutation operator. Consequently, the elaborate detail in *crossover_one* is useful. Our crossover operator employed a few relatively simple graph theoretical properties, yet the implementation was rather involved; even without undertaking the extra task of optimizing its application. We speculate that there exist better performing A-type crossover operators. Furthermore, we speculate that many of these are more complex than *crossover_one*.

- ⋆ We believe that chromosomes for A-type crossover operators can be devised. Furthermore, we speculate that this is a useful thing to do.

- ⋆ With a crossover chromosome one should be able to test whether certain properties (such as the out-degree of nodes, connectedness of subgraphs, network activity*, and perhaps some measure of symmetry) are useful for crossover.

*We use the term *activity* like 'neuron activity' in the brain. Loosely, we can define the activity of a node as the number of changes of state per moment when a large random data packet is processed by that node's

- ⋆ A crossover chromosome would be able to represent many different crossover operators. That is, crossover chromosomes could be used to evolve crossover operators.
- ⋆ For a given problem one could conduct co-evolution of a population of candidate solution A-types and a population of crossover operators for the candidate solutions.
- ⋆ The simplicity of A-types' neurons make A-types an appropriate candidate for researching evolving evolutionary operators for ANNs.

Currently we cannot substantiate the above claims. However, the author maintains that investigating these claims is a worthwhile avenue of research and A-types are a suitable tool for the task.

network. Furthermore, we can define the activity of a subgraph of an A-type as an average of the activity of all nodes in that network. Note that Teuscher [2, ch 5] defines activity of A-types and uses this to investigate the non-linear dynamics of these networks.

# A . Outline of Source Code

In this appendix we give a brief outline of the computer program that we devised and implemented to encode EAs with hypothesis spaces of A-types.

## A.1 Using Linked Objects

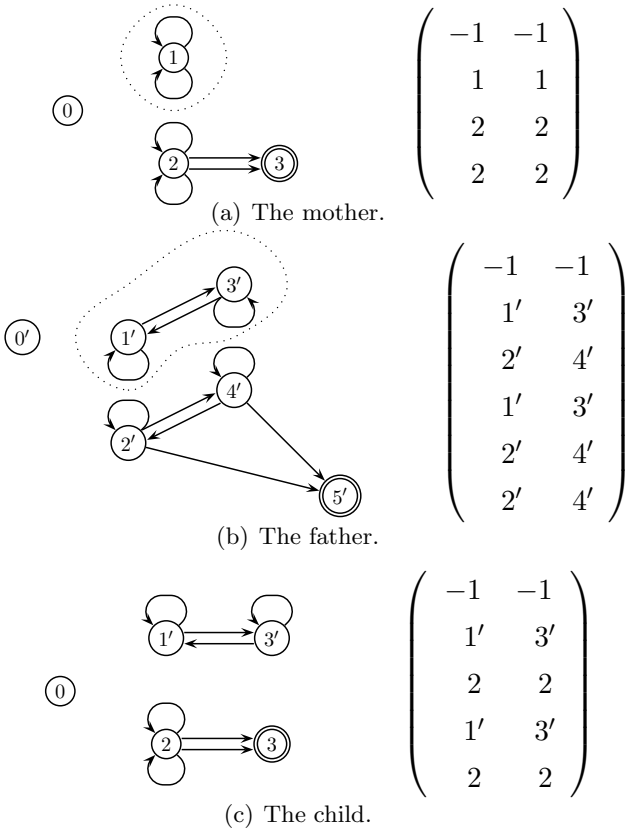
Initially we tried to use adjacency lists to encode A-types. These adjacency lists give linear chromosomes for our A-types*. Our genetic operators were translated into methods that manipulated adjacency lists. Encoding our elaborate crossover method (described in Section 6.6) was very difficult with linear chromosomes. Adjacency lists are constructed from a particular labelling of the parent A-types. For example, consider the crossover illustrated in Figure A.1. Recall that our crossover algorithm composed topologically connected subgraphs from the parents to construct a child. In general, this involves the reconnection of several sections of the parents' adjacency lists. Ensuring that the child's nodes are labelled consistently became rather complex.

We simplify our encoding of crossover by representing A-types in an object-oriented manner. We define classes that represent nodes: we define a class that encodes nand machines and a class that encodes delay machines. We then define an A-type class so that an A-type is an object that references a list of node objects. These node objects can reference one another. An A-type's graph is prescribed by the node references in all of its nodes. Composing subgraphs of two node objects requires the reassignment of references between node objects: no relabelling is required. Using this scheme we successfully encoded *crossover_one* and the 'symmetry' algorithms detailed in Chapter 8. We speculate that this would have been very difficult had we employed adjacency lists.

## A.2 Outline of Source Code

In this section we give a very brief description of the computer program that we constructed (using Java) to represent A-types and our concept learning algorithms. Figure A.2 is a schematic of this program.

*Teuscher [2, p88] used B-types with lists that prescribed whether each connection modifier was in a 'connected' or 'disconnected' state. These lists give a linear chromosomes for Teuscher's A-types.




(a) The mother.

(b) The father.

(c) The child.

**Figure A.1:** Manipulations on adjacency lists can be difficult because relabelling is required. Here we illustrate a simple crossover. This requires no rewiring but it does require relabelling.

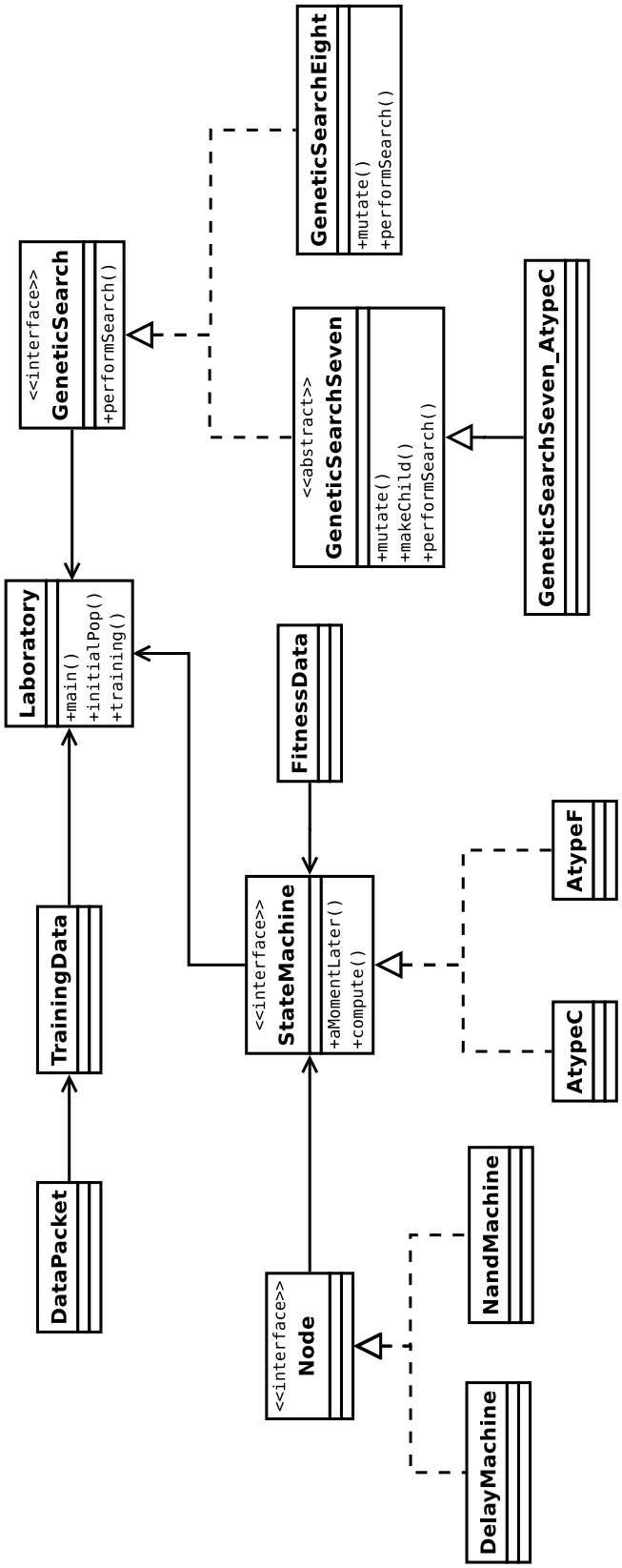



**Figure A.2:** A schematic of our computer program that encodes evolutionary algorithms with hypothesis spaces of A-types.

When designing this program we required flexibility because ideas were being developed in concert with the code development—this seems to almost always be the case. We required a design that allowed us to quickly test new ideas. For example, when we constructed a new algorithm or a new artificial neural network we needed to be able to easily integrate this into the existing code. To do this we used the Template Method Pattern [90, ch 8]. It is from this pattern we designed the StateMachine interface as a template for properties common to all A-types that we encode. Similarly, we designed the GeneticSearch interface as a template for properties common to all of the evolutionary algorithms that we encoded.

Next we give a very brief description the most important classes and methods from our program.

## Class **Laboratory**

This class is where we conduct all of our investigations. This class has the main method. When we conduct an investigation we construct a Laboratory object in which we specify the details of the search. In a Laboratory object we specify the training data, hypothesis space, search algorithm, and initiate and maintain a population of A-types. Next we describe a few of Laboratory's methods.

- ⋆ **main** In Java this method prescribes where the program starts.
- ⋆ **initialPop** This method prescribes how our initial (seed) population is generated for all of our evolutionary algorithms. The type of StateMachines produced depends on the Laboratory.
- ⋆ **training** This method takes a StateMachine and computes the output from all training examples. This method updates the FitnessData object of the input StateMachine.

## Interface **StateMachine**

This interface acts as an template for properties common to all A-types that we encode. All StateMachines maintain a list of Node objects. These Nodes reference one another to prescribe an A-type graph. A StateMachine's state is given by the states of all of its Nodes. Given an input, a StateMachine can determine its next state. In general, these are machines that work in a sequential fashion (recall that clamping is a special case of sequential input) inputting and outputting DataPacket objects. All StateMachines reference a FitnessData object which maintains a list of (delay, fitness) pairs. Because StateMachine is an interface, it defines requirements for its children classes but further detail is provided by these children classes. For instance, AtypeC is a child class of StateMachine and it encodes A-types, yet

AtypeF is also child class of StateMachine and it encodes (A-type, automorphism) pairs. Next we describe a few of StateMachine's methods—note that they are always implemented in the children classes because StateMachine is an Interface rather than a class.

- ⋆ **aMomentLater** This method updates the state of each Node in this StateMachine. If a Node $A$ is a NandMachine object with two source nodes $B$ and $C$ then $A$'s next state is ($B$'s current state) NAND ($C$'s current state). If the node is a DelayMachine object with a source node $D$ then $A$'s next state is the current state of $D$.

- ⋆ **compute** Given an input DataPacket, this method returns an output DataPacket of a specified length and for a specified range of delays.

### Class **FitnessData**

Every StateMachine object has a FitnessData object associated with it. A FitnessData object contains the fitness information for a set of delays for the associated StateMachine. For example, we may want to retain the fitness data for a given StateMachine with delays of 4, 5 and 6 moments. To do this we require a FitnessData object that contains a list of three (delay, fitness) pairs.

## Interface **Node**

This interface acts as a template for our objects that encode nand machines and delay machines. A Node object references two Boolean values: one represents its state at the current moment and the other represents its state at the previous moment—this duplicity simplifies our method of determining the next state of a StateMachine object. A node has a list of references to other nodes that represents incoming arrows. Similarly, a node has a list of references to other nodes that represents outgoing arrows.

## Class **DelayMachine**

This Class is a child class of Node. A DelayMachine object references a single node that represents the incoming arrow.

## Class **NandMachine**

This Class is a child class of Node. A NandMachine object references two nodes that represent the incoming arrows.

## Class **DataPacket**

This class encodes data packets. A DataPacket object has an array of arrays of Boolean variables.

## Class **Training Data**

This class encodes a training set. A TrainingData object has a list of pairs of DataPackets. Each pair encodes an example of the training data.

## Class **AtypeC**[a]

This the child class of StateMachine that encodes A-types.

[a]Note that there are classes AtypeA and AtypeB, etc. These classes proved to be of little use; for instance, AtypeA encoded A-types using adjacency lists.

## Class **AtypeF**

This the child class of StateMachine that encodes (A-type, automorphism) pairs. As required by StateMachine, this class implements a list of Nodes that reference one another and methods that enable AtypeF objects to process DataPackets: this encodes an Atype. In conjunction with this, an AtypeF object has a list of (Node, Node) pairs. This list encodes a $\phi$-table which represents an automorphism.

## Interface **GeneticSearch**

This interface acts as a template for all of our evolutionary algorithms. GeneticSearch prescribes selection rules, fitness functions, mutation operators, and the general procedure for all of our evolutionary searches. GeneticSearch specifies the following method.

- ⋆ **performSearch** This method prescribes the general procedure for all of our evolutionary searches.

## Abstract Class **GeneticSearchSeven**[a]

This class encodes *genetic_search_one*, and its two special cases *blind_search_one* and *mutation_search_one*. Next we describe a few of GeneticSearchSeven's methods.

- ⋆ **mutate** Given an input A-type, this method constructs and returns a new A-type that is a slight variation of the input. This method encodes the algorithm *mutate_one*.
- ⋆ **makeChild** This method uses two input A-types to construct, and return, a new A-type. This method encodes the algorithm *crossover_one*.
- ⋆ **performSearch** This method encodes the algorithm *genetic_search_one*. It employs mutate and makeChild as helper methods.

[a] Again the names are historical: there are classes GeneticSearchOne, GeneticSearchTwo, etc. These proved to be of little use.

## Class **GenSearchSeven_AtypeC**

This implements GeneticSearchSeven with AtypeC objects.

## Class **GeneticSearchEight**

This class encodes our $\phi$-table search. Next we describe a few of GeneticSearchEight's methods.

- ⋆ **addOperator** This method encodes the symmetry operator *add*.
- ⋆ **glueOperator** This method encodes the symmetry operator *glue*.
- ⋆ **splitOperator** This method encodes the symmetry operator *split*.
- ⋆ **rewireOperator** This method encodes the symmetry operator *rewire*.
- ⋆ **deleteOperator** This method encodes the symmetry operator *delete*.
- ⋆ **mutate** Given an input A-type, this method constructs and returns a new A-type that is a slight variation of the input. This method encodes the algorithm *mutate_two*. It employs addOperator, glueOperator, splitOperator, rewireOperator, and deleteOperator as helper methods.
- ⋆ **performSearch** This method is required by the parent class GeneticSearch. In GeneticSearchEight this method encodes the algorithm *mutation_search_one* and this is only used with AtypeF objects. It employs mutate as a helper method.

# Bibliography


[1] A. Turing. Intelligent Machinery: A Report. Technical report, 1948.

[2] C. Teuscher. *Turing's Connectionism: An Investigation of Neural Network Architectures*. Springer, 2001.

[3] C. Teuscher and E. Sanchez. Self-Organizing Topology Evolution of Turing Neural Networks. *Lecture notes in computer science*, pages 820–826, 2001.

[4] M.A. Arbib. *The handbook of brain theory and neural networks*. The MIT Press, 2002.

[5] C.G. Johnson. Genetic Programming Crossover: Does It Cross over? pages 97–108, 2009.

[6] G. Kronberger, S. Winkler, M. Affenzeller, and S. Wagner. On Crossover Success Rate in Genetic Programming with Offspring Selection. pages 232–243, 2009.

[7] D.R. White and S. Poulding. A Rigorous Evaluation of Crossover and Mutation in Genetic Programming. pages 220–231, 2009.

[8] R. I. M. Dunbar. The Social Brain: Mind, Language, and Society in Evolutionary Perspective. *Annual Review of Anthropology*, 32(1):163–181, 2003.

[9] G.F. Miller. *The Mating Mind: How Sexual Choice Shaped the Evolution of Human Nature*. Doubleday, 2000.

[10] M. Hamermesh. *Group theory and its application to physical problems*. Dover Publications, 1989.

[11] R. Herbrich. *Learning kernel classifiers: theory and algorithms*. The MIT Press, 2001.

[12] R. Kondor. *Group Theoretical Methods in Machine Learning*. PhD thesis, Columbia University, 2008.

[13] P. Baldi. Symmetries and learning in neural network models. *Physical review letters*, 59(17):1976–1978, 1987.

[14] J. Shawe-Taylor. Symmetries and discriminability in feedforward network architectures. *IEEE Transactions on Neural Networks*, 4(5):816–826, 1993.

[15] J. Wood and J. Shawe-Taylor. A unifying framework for invariant pattern recognition. *Pattern Recognition Letters*, 17(14):1415–1422, 1996.

[16] J.Y. Dong and J.Y. Zhang. Detection of the permutation symmetry in pattern sets. *Discrete Dynamics in Nature and Society*, 2006, 2006.

[17] L. Džeroski, S. Todorovski, editor. *Computational Discovery of Scientific Knowledge*. Springer-Verlag Berlin, Heidelberg, 2007.

[18] D. Gillies. *Artificial intelligence and scientific method*. Oxford University Press, USA, 1996.

[19] P. Langley, H.A. Simon, G.L. Bradshaw, and J.M. Zytkow. *Scientific discovery: computational explorations of the creative process*. MIT Press Cambridge, MA, USA, 1987.

[20] B. Dolsak and S. Muggleton. The application of Inductive Logic Programming to finite element mesh design. *Inductive logic programming*, pages 453–472, 1992.

[21] R. Mankel. Pattern recognition and event reconstruction. *Reports on Progress in Physics*, 67:553–622, 2004.

[22] R.D. King, K.E. Whelan, F.M. Jones, P.G.K. Reiser, C.H. Bryant, S.H. Muggleton, D.B. Kell, and S.G. Oliver. Functional genomic hypothesis generation and experimentation by a robot scientist. *Nature*, 427(6971):247–252, 2004.

[23] R.D. King, J. Rowland, S.G. Oliver, M. Young, W. Aubrey, E. Byrne, M. Liakata, M. Markham, P. Pir, L.N. Soldatova, et al. The automation of science. *Science*, 324(5923):85, 2009.

[24] J. Rosen. *Symmetry rules: How science and nature are founded on symmetry*. Springer Verlag, 2008.

[25] NK Bose and P. Liang. *Neural network fundamentals with graphs, algorithms, and applications*. McGraw-Hill, Inc. Hightstown, NJ, USA, 1996.

[26] R. Diestel. *Graph Theory*. Springer, 2005.

[27] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *An Introduction to Algorithms*. MIT Press, McGraw-Hill, Massachusetts, 1990.

[28] D.J.S. Robinson. *A Course in the Theory of Groups*. Springer, 1982.

[29] J.L. Gross and T.W. Tucker. *Topological graph theory*. Wiley-Interscience Series in Discrete Mathematics and Optimization, 1987.

[30] D.J.C. MacKay. *Information theory, inference, and learning algorithms*. Cambridge Univiversity Press, 2003.

[31] K.Y. Siu, V.P. Roychowdhury, and T. Kailath. *Discrete neural computation: a theoretical foundation*. Prentice Hall, 1995.

[32] B.G. Buchanan. A (very) brief history of artificial intelligence. *AI Magazine*, 26(4):53, 2005.

[33] M.L. Minsky. *Computation: finite and infinite machines*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1967.

[34] J. Copeland. *Artificial intelligence: A philosophical introduction*. Blackwell Publishing, 1993.

[35] M Kaku. Visions of the future. *BBC Four*, 2007.

[36] H. Markram. The blue brain project. *Nature Reviews Neuroscience*, 7(2):153–160, 2006.

[37] T.L. Manuel. Creating a Robot Culture: An Interview with Luc Steels. *IEEE Intelligent Systems*, 18(3):59 – 61, 2003.

[38] D. Floreano and C. Mattiussi. *Bio-inspired artificial intelligence: Theories, methods, and technologies*. MIT Press, 2008.

[39] A.E. Eiben, P.E. Raué, and Z. Ruttkay. Genetic algorithms with multi-parent recombination. *Lecture Notes in Computer Science*, 866:78–87, 1994.

[40] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, New York, 1997.

[41] R.O. Duda, P.E. Hart, and D.G. Stork. *Pattern classification*. Citeseer, 2001.

[42] S. Haykin. *Neural networks and learning machines*. Prentice Hall, 2008.

[43] W. Banzhaf, P. Nordin, RE Keller, and FD Francone. *Genetic Programming: An Introduction. Morgan Kaufmann*. Morgan Kaufmann, 1998.

[44] KM Saridakis and AJ Dentsoras. Soft computing in engineering design–A review. *Advanced Engineering Informatics*, 22(2):202–221, 2008.

[45] R. Rada. Expert systems and evolutionary computing for financial investing: A review. *Expert Systems with Applications*, 34(4):2232–2240, 2008.

[46] D Fogel, editor. *Evolutionary Computation: the fossil record*. New York : IEEE Press, 1998.

[47] A.E. Eiben and J.E. Smith. *Introduction to evolutionary computing*. springer, 2003.

[48] J.H. Holland. Adaptation in natural and artificial systems. 1975.

[49] J.R. Koza. *Genetic programming: on the programming of computers by means of natural selection*. The MIT press, 1992.

[50] L.J. Fogel, A.J. Owens, and M.J. Walsh. *Artificial intelligence through simulated evolution*. John Wiley & Sons Inc, 1966.

[51] A. Teller and M. Veloso. *Program evolution for data mining*, volume 8, pages 213–236. JAI Press Inc, 1995.

[52] SK Pal, S. Bandyopadhyay, and SS Ray. Evolutionary computation in bioinformatics: A review. *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, 36(5):601–615, 2006.

[53] W.S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biology*, 5(4):115–133, 1943.

[54] M. Strano. A neural network applied to criminal psychological profiling: An Italian initiative. *International Journal of Offender Therapy and Comparative Criminology*, 48(4):495–503, 2004.

[55] M.J. Aitkenhead and A.J.S. McDonald. The state of play in machine/environment interactions. *Artificial Intelligence Review*, 25(3):247–276, 2006.

[56] A.C. Tsoi and A. Back. Discrete time recurrent neural network architectures: A unifying review. *Neurocomputing*, 15(3-4):183–223, 1997.

[57] J. Šíma and P. Orponen. General-purpose computation with neural networks: A survey of complexity theoretic results. *Neural Computation*, 15(12):2727–2778, 2003.

[58] R. Rojas and J. Feldman. *Neural networks: a systematic introduction*. Springer, 1996.

[59] D.M. Eagleman, P.U. Tse, D. Buonomano, P. Janssen, A.C. Nobre, and A.O. Holcombe. Time and the brain: how subjective time relates to neural time. *Journal of Neuroscience*, 25(45):10369, 2005.

[60] D.V. Buonomano and W. Maass. State-dependent computations: spatiotemporal processing in cortical networks. *Nature Reviews Neuroscience*, 10(2):113–125, 2009.

[61] C. Gershenson. Classification of random boolean networks. pages 1–8, 2002.

[62] J.L. Schiff. *Cellular automata: a discrete view of the world*. John Wiley and Sons, 2008.

[63] S.A. Kauffman. *Investigations*. Oxford University Press, USA, 2000.

[64] R. Sun and C.L. Giles. *Sequence Learning: Paradigms, Algorithms, and Applications, number 1828 in Lecture Notes in Artificial Intelligence*. Springer Verlag, 2000.

[65] S. Wolfram. *A new kind of science*. Wolfram Media, Inc, 2002.

[66] M. Gardner. Mathematical games: The fantastic combinations of John Conways new solitaire game Life. *Scientific American*, 223(4):120–123, 1970.

[67] B.J. Copeland and A.M. Turing. *The Essential Turing: Seminal Writings in Computing, Logic, Philosophy, Artificial Intelligence, and Artificial Life Plus the Secrets of Enigma*. Oxford University Press, USA, 2004.

[68] C. Teuscher, editor. *Alan Turing: Life and legacy of a great thinker*. Springer Verlag, 2004.

[69] BJ Copeland and D. Proudfoot. Alan Turing's forgotten ideas in computer science. *Scientific American*, 280(4):98–103, 1999.

[70] B.J. Copeland and D. Proudfoot. On Alan Turing's anticipation of connectionism. *Synthese*, 108(3):361–377, 1996.

[71] M. Alan. Turing. The chemical basis of morphogenesis. *Philosophical Transactions of the Royal Society of London Series B*, 237:37–72, 1952.

[72] T. Rohlf and S. Bornholdt. Self-organized criticality and adaptation in discrete dynamical networks. *Adaptive Networks*, pages 73–106, 2009.

[73] L. Bull. On dynamical genetic programming: Simple boolean networks in learning classifier systems. *International Journal of Parallel, Emergent and Distributed Systems*, 24(5):421–442, 2009.

[74] L. Bull and R. Preen. On Dynamical Genetic Programming: Random Boolean Networks in Learning Classifier Systems. *Lecture Notes in Computer Science*, 5481:37 – 48, 2009.

[75] M.A. Arbib. *Theories of Abstract Automata*. Prentice Hall, 1969.

[76] V. Klenk. *Understanding symbolic logic*. Prentice Hall, 1983.

[77] R. Dawkins. *The Ancestor's Tale: A Pilgrimage to the Dawn of Evolution*. Mariner Books, 2005.

[78] S. Brown and Z. Vranesic. Fundamentals of digital logic with VHDL design. 2004.

[79] Stewart W. Wilson. Classifier systems and the animat problem. *Mach. Learn.*, 2(3):199–228, 1987.

[80] M.V. Butz. *Anticipatory learning classifier systems*. Kluwer Academic Pub, 2002.

[81] Sun Microsystems. Java platform, standard edition 6 api specification. http://java.sun.com/javase/6/docs/api/.

[82] E. Kreyszig. *Advanced Engineering Mathematics*. John Wiley, New York, NY, USA, seventh edition edition, 1994.

[83] T. Jones. Crossover, macromutation, and population-based search. pages 73–80, 1995.

[84] R. Poli and N.F. McPhee. Exact GP Schema Theory for Headless Chicken Crossover with Subtree Mutation. *cognitive science research papers-University of Birmingham CSRP*, 2000.

[85] C.J.S. Webber. Self-Organization of Symmetry Networks: Transformation Invariance from the Spontaneous Symmetry-Breaking Mechanism. *Neural computation*, 12(3):565–596, 2000.

[86] WJ Tang and QH Wu. Biologically inspired optimization: a review. *Transactions of the Institute of Measurement & Control*, pages 495–515, 2009.

[87] A. Teller. Evolving programmers: The co-evolution of intelligent recombination operators. In *Advances in genetic programming*, pages 45–68. MIT Press, 1996.

[88] P. Dittrich, J. Ziegler, and W. Banzhaf. Artificial chemistries-a review. *Artificial Life*, 7(3):225–275, 2001.

[89] C. Teuscher. *Amorphous membrane blending: From regular to irregular cellular computing machines*. PhD thesis, École Polytechnique Fedérale de Lausanne, 2004.

[90] E. Freeman, E. Freeman, B. Bates, and K. Sierra. *Head first design patterns*. O'Reilly & Associates, Inc., 2004.